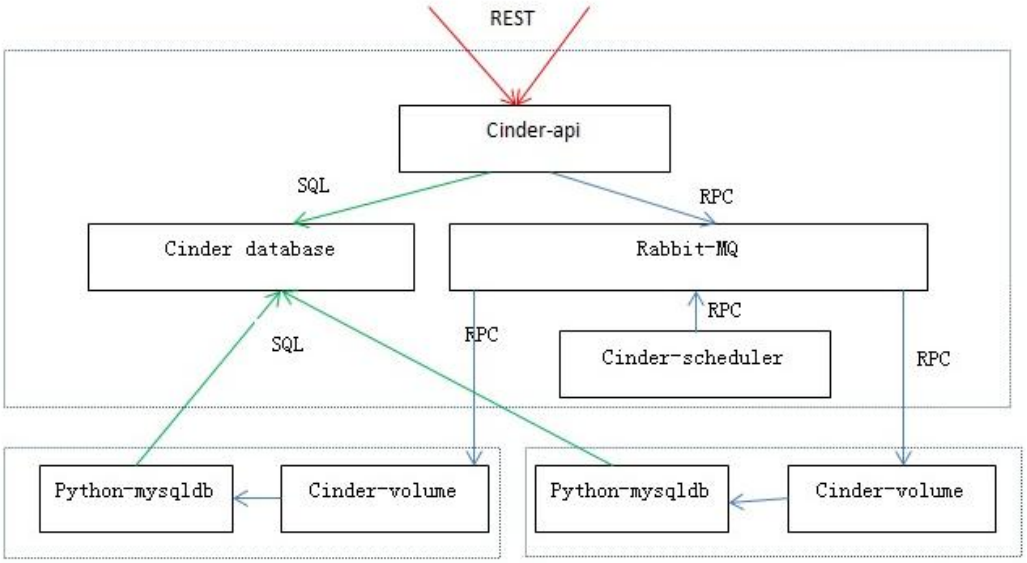
Cinder 代码解析

Cinder 介绍

Cinder 是 openstack 中的块存储解决方案，前身是 nova-volume。Cinder 提供的服务主要有 cinder-volume，cinder-backup，cinder-scheduler，还有一个 cinder-api 服务，并且 Cinder 提供了一个控制台管理工具，提供命令行操作。Cinder-API 负责接收用户请求，进行相应检查，并提取参数为调用具体功能做准备；cinder-scheduler 实现调度功能，选择合适的主机提供云存储，其中 filter 过滤掉不适合的主机，weighter 选择出最合适的主机；cinder-volume 就是实现实际的块存储管理功能，cinder-backup 实现云硬盘的备份功能。



cinder-api 是主要服务接口，负责接受和处理外界的 API 请求，并将请求放入 RabbitMQ 队列，交由后端执行。

cinder-scheduler 的用途是在多 backend 环境中决定新建 volume 的放置 host：

0。 首先判断 host 的状态，只有 service 状态为 up 的 host 才会被考虑。

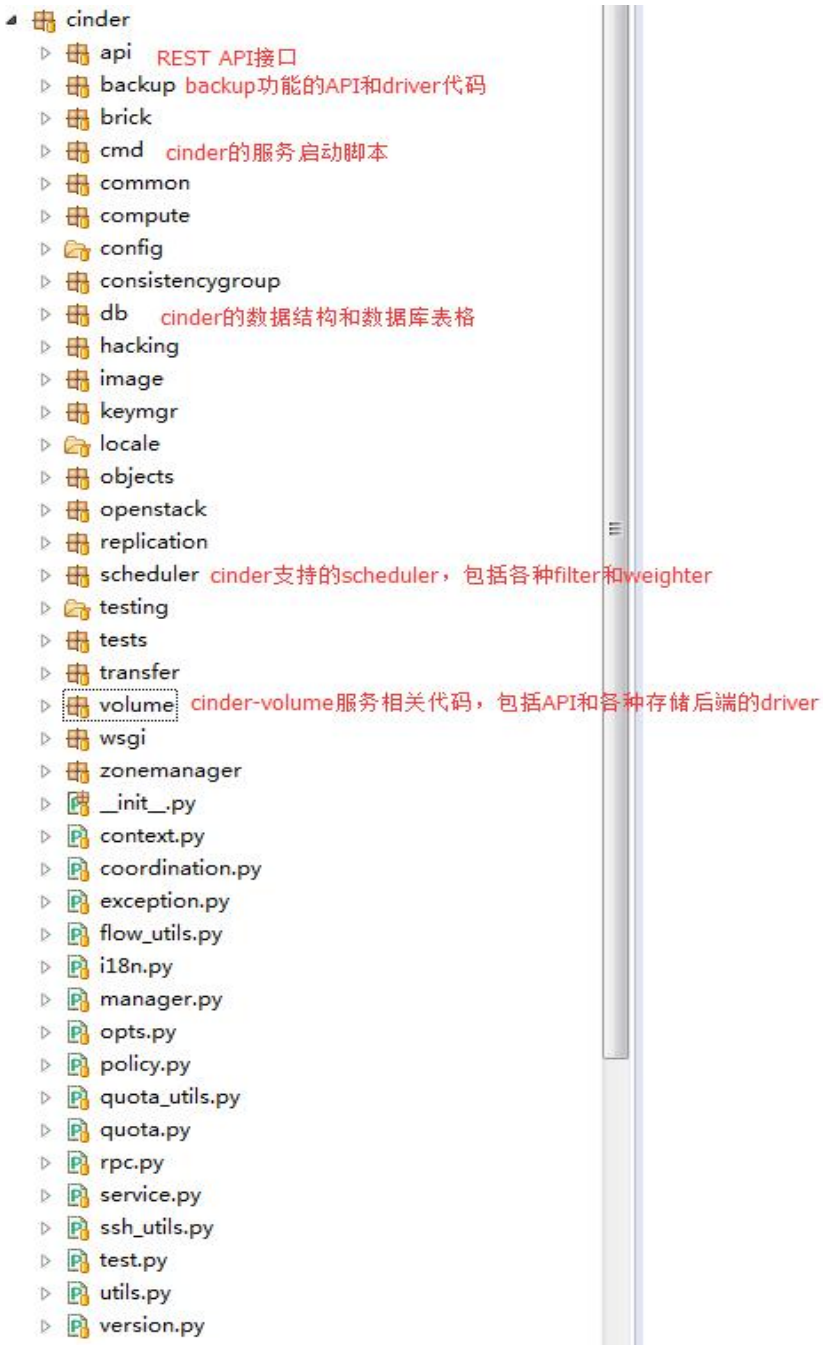
1。创建 volume 的时候，根据 filter 和 weight 算法选出最优的 host 来创建 volume。

2。迁移 volume 的时候，根据 filter 和 weight 算法来判断目的 host 是不是符合要求。

如果选出一个 host，则使用 RPC 调用 cinder-volume 来执行 volume 操作。

cinder-volume 服务运行在存储节点上(对接商业存储时，运行在控制节点上)，管理存储空间，处理 cinder 数据库的维护状态的读写请求，通过消息队列和直接在块存储设备或软件上与其他进程交互。每个存储节点都有一个 Volume Service，若干个这样的存储节点联合起来可以构成一个存储资源池。

Cinder 源码宏观视图



cinderclient 解析

Openstack 各个模块都有客户端的实现，为用户访问具体模块提供了接口，也作为各个模块之间相互访问的途径。Cinder 模块的 client 为 cinderclient，下面以 cinder create 为例，分析源码实现。

Cinder create 命令执行时，首先调用/usr/bin/cinder，根据内容可知实际调用的为/usr/lib/python2.7/site-packages/cinderclient/shell.py 中的 main 函数。

1、函数使用解析器 parser 对参数进行解析，并给响应变量赋值

2、确定使用的 API 版本，并实现解析子命令（create）

3、如有需要，调用 authenticate 对要调用的方法进行权限验证

4、在 main 方法的最后，调用 args.func(self.cs, args)函数，即 do_create 函数

main 函数最终调用了/usr/lib/python2.7/site-packages/cinderclient/v2/shell.py 中的 do_create 函数，即 volume = cs.volumes.create(...)。

cs 则定位了/usr/lib/python2.7/site-packages/cinderclient/v2/client.py 的 class Client（object），而__iniit__中 self.volumes = volumes.VolumeManager(self)。

Volumes 则是/usr/lib/python2.7/site-packages/cinderclient/v2/volumes.py 中的 class VolumeManager(base.ManagerWithFind)类。

故 cs.volume.create()调用的是/usr/lib/python2.7/site-packages/cinderclient/v2/volumes.py 中 VolumeManager 类的 create 函数。

该函数最后一句 self._create('/volumes', body, 'volume')，则是调用了/usr/lib/python2.7/site-packages/cinderclient/base.py 中 Manager 类的_create 函数。

_create 函数的核心语句 resp, body = self.api.client.post(url, body=body)，调用的是/usr/lib/python2.7/site-packages/cinderclient/client.py 中 class HTTPClient 的 post 函数：

```
def post(self, url, **kwargs):
    return self._cs_request(url, 'POST', **kwargs)
```

接着调用该文件内的_cs_request 函数，后者又调用 request 函数

```
def _cs_request(self, url, method, **kwargs):
    auth_attempts = 0
    attempts = 0
    backoff = 1
    while True:
        attempts += 1
        if not self.management_url or not self.auth_token:
            self.authenticate()
        kwargs.setdefault('headers', {})['X-Auth-Token'] = self.auth_token
        if self.projectid:
            kwargs['headers']['X-Auth-Project-Id'] = self.projectid
        try:
            resp, body = self.request(self.management_url + url, method, **kwargs)
```

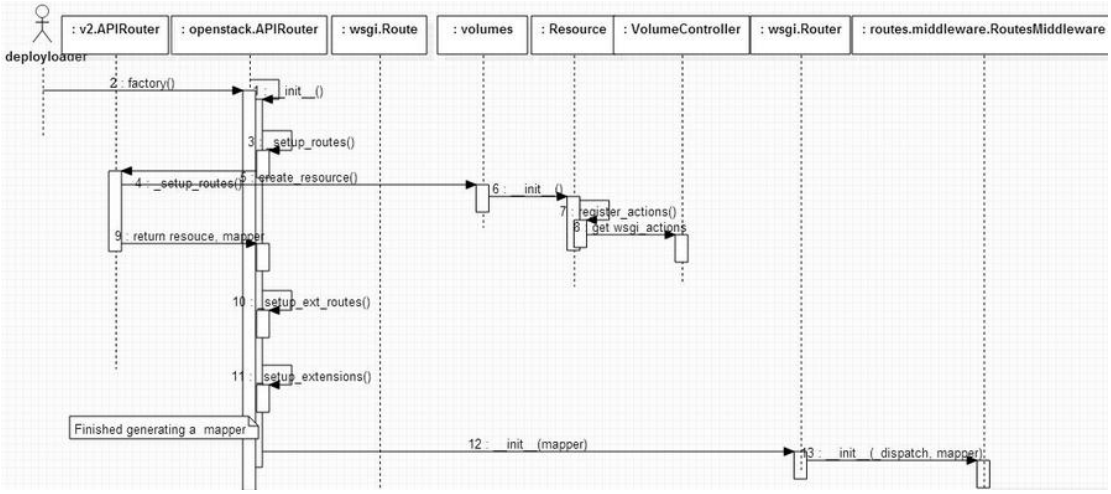
request 函数中调用了 Python 的 request 库函数，实现最终通过 http 协议访问远程服务器，即将请求发给了 server 端。

```
def request(self, url, method, **kwargs):
        kwargs.setdefault('headers', kwargs.get('headers', {}))
        kwargs['headers']['User-Agent'] = self.USER_AGENT
        kwargs['headers']['Accept'] = 'application/json'
        if 'body' in kwargs:
            kwargs['headers']['Content-Type'] = 'application/json'
            kwargs['data'] = json.dumps(kwargs['body'])
            del kwargs['body']

        if self.timeout:
            kwargs.setdefault('timeout', self.timeout)
        self.http_log_req((url, method,), kwargs)
        resp = requests.request(
            method,
            url,
            verify=self.verify_cert,
            **kwargs)
        self.http_log_resp(resp)
```

这里仅仅以 cinder create 为例，分析了 cinderclient 的工作流程，其他 client 的命令流程类似，不再一一解释。

API request 到具体函数的映射



在启动 cinder-api 服务时，根据 paste deploy 机制，会初始化 APIRouter 的实例：

```
def __init__(self, ext_mgr=None):
    if ext_mgr is None:
        if self.ExtensionManager:
            ext_mgr = self.ExtensionManager()
        else:
            raise Exception(_("Must specify an ExtensionManager class"))

    mapper = ProjectMapper()
    self.resources = {}
    self._setup_routes(mapper, ext_mgr)
    self._setup_ext_routes(mapper, ext_mgr)
    self._setup_extensions(ext_mgr)
    super(APIRouter, self).__init__(mapper)
```

1_setup_routes 函数（在/cinder/api/v2/router.py）配置各项核心资源（volume、snapshot 等）：

```
self.resources['volumes'] = volumes.create_resource(ext_mgr)
mapper.resource("volume", "volumes",
                controller=self.resources['volumes'],
                collection={'detail': 'GET'},
                member={'action': 'POST'})
```

1.1 上图第一句，创建一个 resources 实例，记录在 router 的 resources 属性中。create_resource 函数（/cinder/api/v2/volumes.py），先后生成一个 VolumeController 实例和 Resource 实例，VolumeController 实例其实就是管理 volume 相关操作的类。VolumeController 类中有 index、detail、create、update、delete 等函数。以 create 函数为例，经过一系列的赋值和检查，最终调用 cinder/volume/api.py 中的 create 函数

1.2 上图第二句，定义了一个将 URL 映射到 VolumeController method 的规则：

如果 URL 是 /volumes/{volume ID} 并且 HTTP method 是 POST，那么该 URL 会被映射到 action 方法，该 action 会被转化为 VolumeController 的具体方法。

如果 URL 是 /volumes 并且 HTTP Method 是 GET，那么映射到 detail 方法，其对应的是 VolumeConroller 的 detail 方法。

此处，router 的 mapper 已经初始化完成。

后续 server 收到一个 action 后，RoutesMiddleware 根据 URL 找到 Resource 的地址，然后 Resource 从 request body 中取出 action，找到对应的函数进行执行。

创建 volume 代码流程

cinder 模块中实现命令行创建云硬盘操作主要有以下几个步骤：

1.应用若干中间件对客户端发送过来的请求信息进行过滤（封装）处理；

当请求信息过来以后，会先后调用以下类中的__call__方法对其进行处理：

FaultWrapper->RequestBodySizeLimiter->CinderKeystoneContext->APIRouter->Resource

其中：

FaultWrapper：关于错误异常的处理；

RequestBodySizeLimiter：请求信息中 body 部分长度的检测；

CinderKeystoneContext：身份验证相关；

APIRouter：请求信息的路由处理；

Resource：action 执行前的一些预处理；

2.获取要执行的方法及其相关的扩展方法；

获取请求信息中 action 方法的具体路径和获取 action 方法相关的扩展方法及其具体路径

3.请求中 body 部分的反序列化；

获取合适的反序列化方法和应用指定的反序列化方法对 body 进行反序列化操作

4.获取的扩展方法的执行；

5.执行具体的方法，如卷的建立方法 create；

功能就是执行之前获取的 action 方法（cinder.api.v2.volumes. VolumeController.create），获取并返回方法执行的结果，为后续形成方法执行的响应信息做准备。

目前 openstack 的源代码随着模块和功能的增加和完善，代码量快速增长，功能的实现复杂度也随之增加，引进 taskflow 库能够实现方便的代码管理，并且增加功能实现的安全性。简单来说，当实现一个功能时，应用 taskflow 模式能够实现对 flow 执行的管理，能够开始、中止、重新开始以及逆转回滚等操作，

比如当执行某个 flow 操作出现异常时，可以视具体情况尝试进行 flow 的逆转回滚操作，实现回溯到 flow 执行之前的状态。

使用 taskflow 创建云硬盘的过程：

5.1 /cinder/api/v2/volumes.py----class VolumeController----def create

5.1.1 通过 req 和 body 获取建立卷所需的相关参数，因为建立卷的方法有几种，后面会分析到，具体采用哪种卷的建立方法，是根据 req 和 body 中的相关参数决定的，而且卷的建立也许要确定一些参数，例如卷的大小等等；

5.1.2 调用方法 create 实现卷的建立(/cinder/volume/api.py----class API----def create)

5.1.2.1 构建字典 create_what，实现整合建立卷的具体参数；

5.1.2.2 构建并返回用于建立卷的 flow；

5.1.2.3 执行构建的用于建立卷的 flow；依次执行 flow 中的 task，使用_cast_create_volume 创建

5.1.2.3.1 从输入参数 request_spec 中获取相应的数据信息，主要用于鉴别采用何种建立卷的方法；

5.1.2.3.2 判断是否根据现有快照在快照所在的主机上进行新卷的建立，根据判断结果从具体的参数中获取建立新卷的目标主机；

5.1.2.3.3 .如果没有获取到主机信息，说明没有确定建立卷的目标主机，需要通过调度器择优获取目标主机，并实现远程调用方法 create_volume，来进行建立卷的操作；/cinder/scheduler/manager.py----class SchedulerManager----def create_volume

5.1.2.3.3.1 通过构建 taskflow 的方式，先用 filter 过滤掉不合适的 host，然后使用 weighter 找到最合适的 host，以供创建云硬盘，最终也会跳转到 5.1.2.3.4

5.1.2.3.4 如果获取到主机的信息，则直接通过 volume_rpcapi 实现远程调用方法 create_volume，来进行卷的建立操作。/cinder/volume/manager.py----class VolumeManager----def create_volume

5.1.2.3.4.1 根据参数，确定创建 volume 的方式，建立新卷的四种途径，即直接建立 raw 格式的新卷、从快照建立新卷、从已有的卷建立新卷和从镜像建立新卷，然后调用对应 driver 进行实际的 volume 创建。

5.1.2.4 从 flow 中获取建立卷的反馈信息；

5.1.3 获取建立卷后的反馈信息，实现格式转换，并获取其中重要的属性信息，以便上层调用生成卷的建立的响应信息；

6.响应信息的生成；

功能是基于之前获取的 action 方法执行结果，经过填充信息和格式处理，实现形成响应信息

源代码文件概述（按目录）

1 backup（/cinder/backup/）

/cinder/backup/__init__.py：指定并导入 cinder-backup 的 API 类；

/cinder/backup/api.py：处理所有与卷备份服务相关的请求；

class API(base.Base):卷备份管理的接口 API；主要定义了卷的备份相关的三个操作的 API：create：实现卷的备份的建立；delete：实现删除卷的备份；restore：实现恢复备份；这三个操作都需要通过 backup_rpcapi 定义的 RPC 框架类的远程调用来实现；

/cinder/backup/driver.py:所有备份驱动类的基类；

class BackupDriver(base.Base):所有备份驱动类的基类；

/cinder/backup/manager.py：卷备份的管理操作的实现；

class BackupManager(manager.SchedulerDependentManager):块存储设备的备份管理；继承自类SchedulerDependentManager；主要实现的是三个远程调用的方法：create_backup：实现卷的备份的建立（对应 api.py 中的 creat 方法）；restore_backup：实现恢复备份（对应 api.py 中的 restore 方法）；delete_backup：实现删除卷的备份（对应 api.py 中的 delete 方法）；

/cinder/backup/rpcapi.py：volume rpc API 客户端类；

class BackupAPI(cinder.openstack.common.rpc.proxy.RpcProxy):volume rpc API 客户端类，主要实现了三个方法；create_backup：远程调用实现卷的备份的建立（对应 api.py 中的 creat 方法）；restore_backup：远程调用实现恢复备份（对应 api.py 中的 restore 方法）；delete_backup：远程调用实现删除卷的备份（对应 api.py 中的 delete 方法）；

/cinder/backup/drivers/ceph.py：ceph 备份服务实现；

class CephBackupDriver(BackupDriver):Ceph 对象存储的 Cinder 卷备份类；这个类确认备份 Cinder 卷到 Ceph 对象存储系统；

/cinder/backup/drivers/Swift.py：用 swift 作为后端的备份服务的实现；

class SwiftBackupDriver(BackupDriver):用 swift 作为后端的备份服务的各种管理操作实现类；

/cinder/backup/drivers/tsm.py：IBM Tivoli 存储管理（TSM）的备份驱动类；

class TSMBackupDriver(BackupDriver):实现了针对 TSM 驱动的卷备份的备份、恢复和删除等操作；

2 common（/cinder/common）

/cinder/common/config.py：定义和描述了若干配置参数信息；

/cinder/common/sqlalchemyutils.py：sqlalchemy 数据库的实用工具，主要实现了一个方法 paginate_query，这个方法实现了数据库的分页查询；

3 compute（/cinder/compute/）

/cinder/compute/__init__.py：获取和导入 compute API 类，即 cinder.compute.nova.API；

/cinder/compute/aggregate_states.py：描述了主机所有可能状态的集合；

/cinder/compute/nova.py：定义了一个获取 nova 客户端对象的方法，以及通过 nova 客户端实现卷的快照的处理的若干方法；比如建立卷的快照和删除卷的快照的方法；

4 keymgr（/cinder/keymgr/）

/cinder/keymgr/key_mgr.py：密钥管理方法 API；供类 ConfKeyManager 来继承，具体实现管理密钥的方法；

class KeyManager(object):密钥管理接口的基类；

def create_key(self, ctxt, algorithm='AES', length=256, expiration=None,**kwargs):建立一个密钥；

def store_key(self, ctxt, key, expiration=None, **kwargs):存储一个密钥；

def copy_key(self, ctxt, key_id, **kwargs):拷贝一个密钥；

def get_key(self, ctxt, key_id, **kwargs):检索指定的密钥；

def delete_key(self, ctxt, key_id, **kwargs):删除指定的密钥；

/cinder/keymgr/conf_key_mgr.py：从对象配置选项中获取的密钥的管理实现；

classConfKeyManager(key_mgr.KeyManager):密钥管理的实现方法类；

create_key：实现建立密钥；

store_key：实现存储密钥；

copy_key：实现拷贝密钥；

get_key：实现根据指定的 id 值获取相应的密钥；

delete_key：实现删除指定密钥的操作；

def create_key(self, ctxt, **kwargs):实现建立密钥；

def store_key(self, ctxt, key, **kwargs):实现存储密钥；

def copy_key(self, ctxt, key_id, **kwargs):实现拷贝密钥；

def get_key(self, ctxt, key_id, **kwargs):实现根据指定的 id 值获取相应的密钥；

def delete_key(self, ctxt, key_id, **kwargs):实现删除指定密钥的操作；

/cinder/keymgr/key.py：密钥和对称密钥的相关管理类；

class Key(object):表示所有密钥信息的基类；

def get_algorithm(self):获取密钥的算法；

def get_format(self):获取密钥的编码格式；

def get_encoded(self):根据密钥的编码格式返回密钥的格式；

class SymmetricKey(Key):对称密钥的操作类；

def __init__(self, alg, key):建立一个新的对称对象；

def get_algorithm(self):获取对称加密的算法；

def get_format(self):直接返回值'RAW'；

def get_encoded(self):根据编码格式获取密钥；

/cinder/keymgr/not_impiemented_key_mgr.py：密钥管理引发异常的相关实现；

5 transfer（/cinder/transfer/）

/cinder/transfer/api.py：处理所有与转换卷所有权相关的请求；实现卷从一个租户/对象转换到另一个租户/对象；

class API(base.Base):卷之间所有权相互转换操作 API；实现卷从一个租户/对象转换到另一个租户/对象；

def get(self, context, transfer_id):获取卷所有权的转换数据信息；

def delete(self, context, transfer_id):实现删除要转换所有权的卷的相关数据信息；

def get_all(self, context, filters={}):根据具体用户获取能够获取的所有转换所有权的卷的相关转换数据信息；

def _get_random_string(self, length):随机获取指定长度的字符串；

def _get_crypt_hash(self, salt, auth_key):基于字符串 salt 和 auth_key 生成一个随机 hash 值；

def create(self, context, volume_id, display_name):根据给定的信息在卷的转换的数据库表中建立相应的条目；

def accept(self, context, transfer_id, auth_key):接收已经提供转换的卷；

6 image（/cinder/image/）

/cinder/image/glance.py：应用 glance 作为后端的镜像服务的实现；有些操作时通过客户端调用 glance 模块中的相应方法实现的；

def _parse_image_ref(image_href):对 image_href 进行解析；

def _create_glance_client(context,netloc, use_ssl, version=CONF.glance_api_version): 初始化一个新的 glanceclient.Client 对象（Glance 客户端对象）；

class GlanceClientWrapper(object):实现重试操作的 Glance 客户端包装类；主要实现了初始化 glance 客户端对象的方法和通过客户端对象实现调用 glance 模块中的指定方法的方法；

def _create_static_client(self, context, netloc, use_ssl, version):初始化一个新的 glanceclient.Client 对象（Glance 客户端对象）；

def _create_onetime_client(self, context, version):建立一个客户端对象用于一次服务的调用；

def call(self, context, method, *args, **kwargs):调用一个 glance 客户端方法；

class GlanceImageService(object):这个类提供了存储和对磁盘镜像对象的检索等服务；通过 glance 客户端来实现若干方法的调用执行：

detail：获取镜像列表的详细信息，并对相关镜像的元数据中的属性进行一定的格式转化操作；

show：根据给定的镜像 id 值，返回包含镜像数据的字典；

get_location：获取表示镜像后端存储位置的直接 URL，获取信息：(image_meta.direct_url,image_meta.locations)；

download：通过 glance 客户端实现调用 glance 模块中对应的 data 方法，实现下载指定的镜像数据；

create：存储镜像数据并返回新的镜像对象；

update：应用新的数据更新指定镜像，通过客户端调用 glance 模块中的 upload 方法实现指定镜像元数据的更新；

delete：通过客户端调用 glance 模块中的 delete 方法实现删除指定镜像的操作；

def detail(self, context, **kwargs):获取镜像列表的详细信息，并对相关镜像的元数据中的属性进行一定的格式转化操作；

def _extract_query_params(self, params):获取查询参数；

def show(self, context, image_id):根据给定的镜像 id 值，返回包含镜像数据的字典；

def get_location(self, context, image_id):获取表示镜像后端存储位置的直接 URL；获取信息：(image_meta.direct_url, image_meta.locations)；

def download(self, context, image_id, data=None):通过 glance 客户端实现调用 glance 模块中对应的 data 方法，实现下载指定的镜像数据；

def create(self, context, image_meta, data=None):存储镜像数据并返回新的镜像对象；

def update(self, context, image_id, image_meta, data=None,purge_props=True):应用新的数据更新指定镜像；通过客户端调用 glance 模块中的 upload 方法实现指定镜像元数据的更新；

def delete(self, context, image_id):通过客户端调用 glance 模块 delete 方法实现删除指定镜像的操作；

def _translate_from_glance(image):将镜像元数据中的相关属性值转换为一定的格式；

def _is_image_available(context, image):检测指定镜像是否存在；

def_convert_timestamps_to_datetimes(image_meta):从 ISO 8601 格式的时间戳中解析时间信息，包括镜像建立时间、镜像更新时间和镜像删除时间等；将解析后的时间信息替换镜像元数据中的对应时间戳属性，并返回更新后的镜像元数据；

def _extract_attributes(image):从镜像中提取属性信息；

defget_remote_image_service(context, image_href):建立一个 image_service 并从给定的 image_href 中解

析出 id 值；

def get_default_image_service():获取默认的镜像服务类；

/cinder/image/image_utils.py：执行镜像处理操作的辅助方法；

7 scheduler（/cinder/scheduler/）

/cinder/scheduler/driver.py：所有调度器类都应该继承的调度器基类；

class Scheduler(object):调度器基类；

def get_host_list(self):从 HostManager 获取主机列表；#注：这个方法目前还没有进行具体应用；

def get_service_capabilities(self):从 HostManager 获取服务的 capabilities；#注：这个方法目前还没有进行具体应用；

def update_service_capabilities(self, service_name, host, capabilities):更新服务节点的 capability 信息；

def hosts_up(self, context, topic):获取所有运行了给定主题的主机列表；

def host_passes_filters(self, context, volume_id, host,filter_properties):检测主机是否通过过滤器；

/cinder/scheduler/chance.py：随即选取节点调度器；

classChanceScheduler(driver.Scheduler):随即选取节点调度器；随机选取节点调度器的实现；

def _filter_hosts(self, request_spec, hosts, **kwargs):基于 request_spec 实现对主机列表的过滤；

def _get_weighted_candidates(self, context, topic, request_spec,**kwargs):获取经过 request_spec 过滤的可用的主机列表；

def _schedule(self, context, topic, request_spec, **kwargs):实现随机选取主机；

def schedule_create_volume(self, context, request_spec,filter_properties):实现随机选取主机；远程调用实现在选取的主机上建立并导出卷；

def host_passes_filters(self, context, host, request_spec,filter_properties):检测指定的 host 是否通过了过滤器的过滤；

/cinder/scheduler/filter_scheduler.py：FilterScheduler 用于建立卷选取目标主机；可以应用这个调度器来指定我们要应用的卷过滤器和称重方法来实现对候选主机的过滤和称重操作；

classFilterScheduler(driver.Scheduler):用于对主机进行过滤和称重操作的调度器；

def schedule(self, context, topic, method, *args, **kwargs):对主机进行过滤称重操作获取最优主机；

def _get_configuration_options(self):获取配置选项；

def populate_filter_properties(self, request_spec, filter_properties):填充过滤器属性信息；

def schedule_create_volume(self, context, request_spec, filter_properties):对主机进行过滤和称重操作，获取最优主机，并实现远程调用建立并导出卷；

def host_passes_filters(self, context, host, request_spec,filter_properties):检测指定主机是否经过了主机过滤和称重操作，即在经过主机过滤和称重操作所获取的主机列表中查找是否有指定的主机；

def _post_select_populate_filter_properties(self, filter_properties,host_state):在最优主机选定之后，添加附加信息到过滤器属性；

def _add_retry_host(self, filter_properties, host):增加 retry 条目属性到卷的后端，当响应请求进行主机列表的重新选取时，这个条目属性将会表示某主机是否已经进行过过滤操作，如果已经进行过过滤操作，则不再进行过滤操作，这也是提高系统效率的一个措施；

def _max_attempts(self):获取调度一个卷重试次数的最大值；

def _log_volume_error(self, volume_id, retry):记录卷操作的错误，以辅助代码开发和调试；

def _populate_retry(self, filter_properties, properties):添加或更新重试次数属性值到过滤器属性中；

def _get_weighted_candidates(self, context, request_spec,filter_properties=None): 返回满足request_spec 要求的主机列表，并对主机进行称重操作；

def _schedule(self, context, request_spec, filter_properties=None):对主机进行过滤称重操作，获取最优的主机；

/cinder/scheduler/host_manager.py：管理当前 zone 的 hosts；

class HostState(object):更新主机的可变和不可变的信息；

def update_capabilities(self, capabilities=None, service=None):用给定的 capabilities 和 service，更新现有的 capabilities 和 service；

def update_from_volume_capability(self, capability):从 volume_node 信息 capability 中获取相关数据信息来更新对应的主机信息；

def consume_from_volume(self, volume):实现从卷的信息更新主机状态；

class HostManager(object):主机管理基类；

def _choose_host_filters(self, filter_cls_names):实现检测默认使用的过滤器类是否都可以使用，获取系统默认使用的过滤器类中通过验证的类的列表；

def _choose_host_weighers(self, weight_cls_names):实现检测默认使用的称重类是否都可以使用，获取系统默认使用的称重类中通过验证的类的列表；

def get_filtered_hosts(self, hosts, filter_properties,filter_class_names=None):根据所有过滤器对主机进行过滤操作，只返回符合条件的主机列表；

def get_weighed_hosts(self, hosts, weight_properties,weigher_class_names=None):对主机进行称重操作，返回排序后的主机列表；

def update_service_capabilities(self, service_name, host, capabilities):根据通知更新每个服务的 capabilities；

def get_all_host_states(self, context):获取所有匹配的主机

/cinder/scheduler/manager.py：调度器管理服务；

classSchedulerManager(manager.Manager):选取一个 host 来建立 volumes；

def get_host_list(self, context):从 HostManager 获取主机列表；#注：这个方法目前还没有进行具体应用；

def get_service_capabilities(self, context):从 HostManager 获取服务的 capabilities；#注：这个方法目前还没有进行具体应用；

def update_service_capabilities(self, context, service_name=None,host=None, capabilities=None, **kwargs):更新一个服务节点的 capability 信息；

def create_volume(self, context, topic, volume_id, snapshot_id=None,image_id=None, request_spec=None,filter_properties=None):volume_rpcapi.create_volume 是调用 manager.py 中的 create_volume 方法，该方法先从数据库中取出 volume 的信息，然后调用 volume_utils.notify_about_volume_usage 方法(是不是通知 RabbitMQ?)。然后继续从 volume 信息中取 vol_name，vol_size，并且如果入参中有 snapshot_id，说明从快照创建卷，则从 DB 中取该 snapshot 的信息，如果入参中有 source_volID，说明是从已有卷创建卷，则从 DB 中取该源卷信息，如果入参中有 image_id，说明是从镜像创建卷，则从 glance 中获取镜像服务器信息，镜像 ID，镜像位置，镜像的 metadata.然后调用该类的私有方法_create_volume,该方法首先判断如果 snapshot_ref，imag_id,srcvol_ref 都是空，则说明是创建一个空卷，就调用 driver.create_volume 去创卷，如果有 snapshot_ref 参数，则调用 driver.create_volume_from_snapshot 方法去创卷，如果请求中有源卷的参数，则调用 driver.create_cloned_volume 去创卷(实际上就是克隆一个卷)。如果请求中有镜像参数，则调用 driver.clone_image 方法去创卷，如果 clone_image 失败，则调用普通创卷方法先创建个空卷，然后将卷状态置为 downloading,然后调用_copy_image_to_volume 方法把镜像内容拷贝入卷中。

def request_service_capabilities(self, context):远程调用实现收集驱动的状态和 capabilities 信息，并进行信息的发布；

def migrate_volume_to_host(self, context, topic, volume_id, host,force_host_copy, request_spec, filter_properties=None):确认 host 的存在性并接收指定的卷；即先确定指定的主机是否经过了过滤操作，如果经过了过滤操作，将其作为目标主机，将指定的卷迁移到目标主机上；

def _set_volume_state_and_notify(self, method, updates, context, ex,request_spec):设置卷的状态信息，并进行通知操作；

/cinder/scheduler/rpcapi.py：scheduler 管理 RPC API 的客户端；

classSchedulerAPI(cinder.openstack.common.rpc.proxy.RpcProxy):scheduler 管理 RPC API 的客户端；

def create_volume(self, ctxt, topic, volume_id, snapshot_id=None,image_id=None, request_spec=None, filter_properties=None):远程调用实现卷的建立操作；

def migrate_volume_to_host(self, ctxt, topic, volume_id, host,force_host_copy=False, request_spec=None, filter_properties=None):远程调用实现确认 host 的存在性并接收指定的卷；

def update_service_capabilities(self, ctxt, service_name, host,capabilities):远程调用实现获取 capabilities 信息；

/cinder/scheduler/scheduler_options.py：检测 json 文件的变化，如果需要要进行文件的加载；

class SchedulerOptions(object):检测 json 文件的变化，如果需要要进行文件的加载；

/cinder/scheduler/simple.py：简单调度器；选取拥有最少卷的主机作为目标主机；

classSimpleScheduler(chance.ChanceScheduler):简单的主机调度器类；

def schedule_create_volume(self, context, request_spec,filter_properties):选取拥有最少卷的主机作为目标主机；

/cinder/scheduler/filters/capacity_filter.py：基于卷主机 capacity 使用率的 CapacityFilter 过滤器；

classCapacityFilter(filters.BaseHostFilter):基于卷主机 capacity 使用率的 CapacityFilter 过滤器；

def host_passes(self, host_state, filter_properties):如果主机有足够的 capacity，返回 True；根据过滤器参数 filter_properties.get('size')对主机进行过滤；

/cinder/scheduler/filters/retry_filter.py：过滤掉那些已经尝试过的节点；

classRetryFilter(filters.BaseHostFilter):过滤掉那些已经尝试过的节点；

def host_passes(self, host_state, filter_properties):跳过那些已经尝试过的节点；

/cinder/scheduler/weights/capacity.py：Capacity 称重；根据主机可用的 available 来进行称重主机的操作；

classCapacityWeigher(weights.BaseHostWeigher):Capacity 称重；根据主机可用的 available 来进行称重主机的操作；

8 taskflow（/cinder/taskflow/）

/cinder/taskflow/decorators.py：flow 的包装管理相关方法；

/cinder/taskflow/exceptions.py：针对 taskflow 库的异常处理类；

/cinder/taskflow/states.py：表示 Job 状态和 Flow 状态；

/cinder/taskflow/task.py：这里定义了 task 的抽象的概念；

class Task(object):这里定义了 task 的抽象的概念；

def __call__(self, context, *args, **kwargs):实现调用类的方法；

def revert(self, context, result, cause):task 任务的逆转回滚方法；

/cinder/taskflow/utils.py：Flow 管理相关的实用工具；

/cinder/taskflow/patterns/base.py：Flow 抽象类；

class Flow(object):Flow 抽象类；

def name(self):flow 可读的非唯一的名称；

def uuid(self):flow 唯一的标识；

def state(self):为 flow 提供了一个只读的状态信息；

def _change_state(self, context, new_state):改变目前 flow 状态为新状态 new_state 并执行通知操作；

def add(self, task):添加一个给定的 task 到工作流中；

def add_many(self, tasks):添加给定的若干的 task 到工作流中；

def interrupt(self):尝试中断当前的 flow 和当前没有在 flow 中执行的 task；

def reset(self):完全重置 flow 的内部的状态，并允许 flow 再次运行；

def soft_reset(self):部分地重置 flow 的内部状态，并允许 flow 从中止的状态再次运行；

def run(self, context, *args, **kwargs):工作流（workflow）的执行操作；

def rollback(self, context, cause):执行 workflow 和其父 workflow 的回滚操作；

/cinder/taskflow/patterns/linear_flow.py：线性工作流管理类；

class Flow(base.Flow):继承自类 Flow；线性工作流管理类；

def add(self, task):添加一个给定的 task 到 flow；

def remove(self, uuid):删除 flow 中 uuid 指定的任务；

def run(self, context, *args, **kwargs):工作流（workflow）的执行操作；

def reset(self):完全重置 flow 的内部的状态，并允许 flow 再次运行；

def rollback(self, context, cause):执行 workflow 和其父 workflow 的回滚操作；

9 volume（/cinder/volume/）

/cinder/volume/__init__.py：定义了所使用的卷 API 类；

/cinder/volume/api.py：处理卷相关的所有请求；

class API(base.Base):卷的管理操作接口 API；

def list_availability_zones(self):描述已知可用的 zone；

def create(self, context, size, name, description, snapshot=None,image_id=None, volume_type=None, metadata=None, availability_zone=None,source_volume=None, scheduler_hints=None, backup_source_volume=None):实现建立卷的操作；

def delete(self, context, volume, force=False):实现卷的删除操作；

def update(self, context, volume, fields):在卷上设置给定的属性，并进行更新；

def get(self, context, volume_id):根据 volume_id 获取相应的 volume；

def get_all(self, context, marker=None, limit=None,sort_key='created_at', sort_dir='desc', filters={}):获取所有卷的信息；

def get_snapshot(self, context, snapshot_id):获取指定卷的快照；

def get_volume(self, context, volume_id):根据 volume_id 获取 volume；

def get_all_snapshots(self, context, search_opts=None):获取属于指定上下文环境中用户的所有的卷的快照；

def reserve_volume(self, context, volume):卷信息的预留保存；

def attach(self, context, volume, instance_uuid, host_name, mountpoint,mode):实现卷的附加操作；

def detach(self, context, volume):实现卷的卸载操作；

def initialize_connection(self, context, volume, connector):初始化卷的连接操作；

def terminate_connection(self, context, volume, connector, force=False):通过连接器从主机清理连接；

def accept_transfer(self, context, volume, new_user, new_project):实现存储器上卷的所有权的转换；指定了要转换所有权的卷 volume、新的用户 new_user 和新的对象 new_project；

def _create_snapshot(self, context, volume, name, description,force=False, metadata=None):实现建立并导出快照；

def create_snapshot(self, context, volume, name, description,metadata=None):实现建立并导出快照；

def create_snapshot_force(self, context, volume, name, description,metadata=None):实现建立并导出快照；

def delete_snapshot(self, context, snapshot, force=False):实现删除快照；

def update_snapshot(self, context, snapshot, fields):为一个快照设置给定属性并进行更新；

def get_volume_metadata(self, context, volume):获取卷相关联的所有元数据；

def delete_volume_metadata(self, context, volume, key):删除卷的元数据序列；

def _check_metadata_properties(self, context, metadata=None):检测卷元数据的属性；

def update_volume_metadata(self, context, volume, metadata,delete=False):更新或建立卷的元数据；

def get_volume_metadata_value(self, volume, key):获取卷的元数据中 key 对应的值；

def get_volume_admin_metadata(self, context, volume):管理员用户获取指定 volume_id 的卷的元数据信息；

def delete_volume_admin_metadata(self, context, volume, key):删除给定卷的元数据项；

def update_volume_admin_metadata(self, context, volume, metadata,delete=False):更新或建立卷的管理元数据；

def get_snapshot_metadata(self, context, snapshot):获取一个快照的所有相关联的元数据；

def delete_snapshot_metadata(self, context, snapshot, key):在数据库中删除给定快照的元数据序列；

def update_snapshot_metadata(self, context, snapshot, metadata,delete=False):如果数据库中指定快照的元数据存在，则进行更新，如果快照的元数据不存在，则建立相应的条目信息；

def get_volume_image_metadata(self, context, volume):获取指定卷的 glance 中的元数据；

def_check_volume_availability(self, context, volume, force):检测卷是可用的；

def copy_volume_to_image(self, context, volume, metadata, force):为指定的卷建立一个新的镜像，并实现上传指定的卷到 Glance；

def extend(self, context, volume, new_size):调用存储后端 extend_volume 方法实现卷大小扩展操作；

def migrate_volume(self, context, volume, host, force_host_copy): 调用存储后端的 migrate_volume_to_host 方法，实现迁移卷到指定的主机；

def migrate_volume_completion(self, context, volume, new_volume, error): 调用存储后端的 migrate_volume_completion 方法，实现卷的迁移的完成操作；

/cinder/volume/configuration.py：为所有卷的驱动器提供配置支持；

/cinder/volume/driver.py：卷的驱动类；

class VolumeDriver(object):执行和存储卷相关的命令；

def get_version(self):获取驱动器当前的版本信息；

def create_volume(self, volume):卷的建立；

def create_volume_from_snapshot(self, volume, snapshot):根据快照建立卷；

defcreate_cloned_volume(self, volume, src_vref):创建指定卷的克隆；

def delete_volume(self, volume):卷的删除；

def create_snapshot(self, snapshot):快照的建立；

def delete_snapshot(self, snapshot):快照的删除；

def create_export(self, context, volume):创建 target，并将指定的逻辑卷加入 LUN；

def remove_export(self, context, volume):删除卷的导出；

def initialize_connection(self, volume, connector):允许连接到连接器并返回连接信息；

def terminate_connection(self, volume, connector, **kwargs):断开从连接器的连接；

def attach_volume(self, context, volume, instance_uuid, host_name,mountpoint):附加到实例或主机的卷的回调方法；

def detach_volume(self, context, volume):卷的卸载的回调方法；

def get_volume_stats(self, refresh=False):返回卷的服务的当前状态；

def do_setup(self, context):初始化 volume driver 的操作；

def validate_connector(self, connector):如果连接器不包含驱动所需的所有数据，则结果为 fail；

def _copy_volume_data_cleanup(self, context, volume, properties,attach_info, remote, force=False):实现从主机断开卷的连接，并清理连接的数据信息；

def copy_volume_data(self, context, src_vol, dest_vol, remote=None):从 src_vol 到 dest_vol 复制数据；

def copy_image_to_volume(self, context, volume, image_service,image_id):从 image_service 获取镜像数据并写入卷；

def copy_volume_to_image(self, context, volume, image_service,image_meta):拷贝卷到指定的镜像；

def _attach_volume(self, context, volume, properties, remote=False):实现附加一个卷；

def _detach_volume(self, attach_info):从主机断开卷的连接；

def clone_image(self, volume, image_location, image_id):从现有的镜像有效的建立一个卷；

def backup_volume(self, context, backup, backup_service):为一个已存在的卷创建新的备份；

def restore_backup(self, context, backup, volume, backup_service):恢复一个备份到一个新的或者是存在的卷；

def clear_download(self, context, volume):中断一个镜像的拷贝之后清理缓存信息；

def migrate_volume(self, context, volume, host):迁移卷到指定的主机；

class ISCSIDriver(VolumeDriver):执行与 ISCSI 卷相关的命令；执行支持 ISCSI 协议的存储卷相关的命令；

class ISERDriver(ISCSIDriver):执行与 ISER 卷相关的命令；

class FibreChannelDriver(VolumeDriver):执行 Fibre Channel volumes 的相关命令；

/cinder/volume/manager.py：卷的管理，管理卷的建立、挂载、卸载以及持久性存储等功能；

class VolumeManager(manager.SchedulerDependentManager):管理可连接的块存储设备；

def create_volume(self, context, volume_id, request_spec=None,filter_properties=None, allow_reschedule=True, snapshot_id=None, image_id=None,source_volid=None):建立并导出卷；

def delete_volume(self, context, volume_id):删除卷的操作；

def create_snapshot(self, context, volume_id, snapshot_id):调用存储后端的 create_snapshot 方法，实现调用实现建立并导出快照；

def delete_snapshot(self, context, snapshot_id):调用存储后端 delete_snapshot 方法，实现删除快照；

def attach_volume(self, context, volume_id, instance_uuid, host_name,mountpoint, mode):调用具体存储后端的 attach_volume 方法，实现卷的附加操作，并更新数据库表明卷已经附加；

def detach_volume(self, context, volume_id):调用存储后端的 detach_volume 方法，实现卷的卸载操作，并更新数据库来表明卷已经卸载；

def copy_volume_to_image(self, context, volume_id, image_meta):调用存储后端的 copy_volume_to_image 方法，实现上传指定的卷到 Glance；

def initialize_connection(self, context, volume_id, connector):调用存储后端的 initialize_connection 方法，实现初始化卷的连接操作；

def terminate_connection(self, context, volume_id, connector,force=False):调用存储后端的 terminate_connection 方法，实现通过连接器从主机清理连接；

def accept_transfer(self, context, volume_id, new_user, new_project):调用存储后端的 accept_transfer 方法，实现存储器上卷的所有权的转换；指定了要转换所有权的卷 volume、新的用户 new_user 和新的对象 new_project；

def _migrate_volume_generic(self, ctxt, volume, host):普通的卷的迁移方法；

def migrate_volume_completion(self, ctxt, volume_id, new_volume_id,error=False):卷的迁移结束之后的操作；

def migrate_volume(self, ctxt, volume_id, host, force_host_copy=False):迁移卷到指定的主机；

def _report_driver_status(self, context):获取卷的状态信息；

def publish_service_capabilities(self, context):收集驱动状态和 capabilities 信息，并进行信息的发布；

def _notify_about_volume_usage(self, context, volume, event_suffix,extra_usage_info=None):获取指定卷的使用率信息，并进行通知操作；

def _notify_about_snapshot_usage(self, context, snapshot, event_suffix,extra_usage_info=None):从卷的快照获取卷的使用率信息，并进行通知操作；

def extend_volume(self, context, volume_id, new_size):卷的扩展；调用存储后端的 extend_volume 方法，实现卷大小的扩展操作；

/cinder/volume/qos_specs.py：QoS 功能的实现；QoS（Quality of Service）服务质量，是网络的一种安全机制；是用来解决网络延迟和阻塞等问题的一种技术；在正常情况下，如果网络只用于特定的无时间限制的应用系统，并不需要 QoS；比如 Web 应用，或 E-mail 设置等；但是对关键应用和多媒体应用就十分必要；当网络过载或拥塞时，QoS 能确保重要业务量不受延迟或丢弃，同时保证网络的高效运行；

def create(context, name,specs=None):在数据库中建立一个 qos_specs；

def update(context, qos_specs_id,specs):更新数据库中 QOS 功能的数据信息；

def delete(context, qos_specs_id,force=False):标志 QOS 功能为删除标识；

def delete_keys(context,qos_specs_id, keys):设置指定的目标 QOS 功能的标识为 delete 标识；

def get_associations(context,specs_id):根据给定的 qos_specs 的 id 值获取所有相关的卷的类型信息；

def associate_qos_with_type(context, specs_id, type_id):根据给定的卷的类型解除相关的 qos_specs；

def disassociate_qos_specs(context,specs_id, type_id):解除卷类型从指定的 qos_specs；

def disassociate_all(context,specs_id):从所有的实体中消除与 specs_id 相关联的 qos_specs；

def get_all_specs(context,inactive=False, search_opts={}):获取所有符合条件的 qos_specs；

def get_qos_specs(ctxt, id):根据给定的 id 值获取单个的 QOS 功能；

def get_qos_specs_by_name(context,name):根据给定的名称获取单个 QOS 功能的相关信息；

/cinder/volume/rpcapi.py：volume RPC API 客户端；

class VolumeAPI(cinder.openstack.common.rpc.proxy.RpcProxy):volume RPC API 客户端；

def create_volume(self, ctxt, volume, host, request_spec,filter_properties, allow_reschedule=True, snapshot_id=None, image_id=None,source_volid=None):远程调用实现建立并导出卷；

def delete_volume(self, ctxt, volume):远程调用实现卷的删除；

def create_snapshot(self, ctxt, volume, snapshot):调用存储后端的 create_snapshot 方法，实现远程调用实现建立并导出快照；

def delete_snapshot(self, ctxt, snapshot, host):调用存储后端的 delete_snapshot 方法，实现远程调用实现删除快照；

def attach_volume(self, ctxt, volume, instance_uuid, host_name,mountpoint, mode):远程调用实现卷的附加操作；调用具体存储后端的 attach_volume 方法，实现卷的附加操作，并更新数据库表明卷已经附加；

def detach_volume(self, ctxt, volume):远程调用实现卷的卸载操作；

def copy_volume_to_image(self, ctxt, volume, image_meta):远程调用实现上传指定的卷到 Glance；

def initialize_connection(self, ctxt, volume, connector):远程调用实现初始化卷的连接操作；调用存储

后端的 initialize_connection 方法，实现初始化卷的连接操作；

def terminate_connection(self, ctxt, volume, connector, force=False):远程调用实现通过连接器从主机清理连接；调用存储后端的 terminate_connection 方法，实现通过连接器从主机清理连接；

def publish_service_capabilities(self, ctxt):远程调用实现收集驱动的状态和 capabilities 信息，并进行信息的发布；

def accept_transfer(self, ctxt, volume, new_user, new_project):调用存储后端的 accept_transfer 方法，实现存储器上卷的所有权的转换；指定了要转换所有权的卷 volume、新的用户 new_user 和新的对象 new_project；

def extend_volume(self, ctxt, volume, new_size):远程调用实现卷大小的扩展操作；

def migrate_volume(self, ctxt, volume, dest_host, force_host_copy):远程调用实现迁移卷到指定主机；

def migrate_volume_completion(self, ctxt, volume, new_volume, error):远程调用实现卷的迁移结束之后的操作；

/cinder/volume/utils.py：卷相关的实用工具和方法；

/cinder/volume/volume_types.py：内置卷的类型属性相关方法；

def create(context, name,extra_specs={}):在数据库中建立一个新的卷的类型；

def destroy(context, id):在数据库中删除卷的类型信息；

def get_all_types(context,inactive=0, search_opts={}):获取所有数据库中没有删除的卷的类型；

def get_volume_type(ctxt, id):根据给定 id 来检索获取单个的卷的类型；

def get_volume_type_by_name(context, name):根据名称获取单个卷的类型；

def get_default_volume_type():获取默认的卷的类型；

def is_encrypted(context,volume_type_id):验证卷的类型是否是加密的；

def get_volume_type_qos_specs(volume_type_id):根据给定的卷类型获取所有 QOS 功能相关的信息；

/cinder/volume/flows/base.py：flow 的基类实现；

def _make_task_name(cls,addons=None):获取任务类的名称；

class CinderTask(task.Task):所有 cinder 任务的基类；

class InjectTask(CinderTask):这个类实现了注入字典信息到 flow 中；

/cinder/volume/flows/utils.py：flow 相关的一些实用工具；

/cinder/volume/flows/creat_volume/__init__.py：基于 flow task 管理方式的卷的建立的实现；

class ExtractVolumeRequestTask(base.CinderTask):实现提取并验证处理卷的请求信息任务类；这个 task 的主要任务是提取和验证输入的值，这些输入的值将形成一个潜在的卷的请求信息；并且实现根据一组条件对这些输入值进行验证，并将这些输入值转换成有效的集合；并返回这些经过验证和转换的输入值，这些输入值将会应用于其他 task 中；

def _extract_snapshot(snapshot):从给定的快照中提取快照的 id 信息；

def _extract_source_volume(source_volume):从给定的卷中提取卷的 id 信息；

def _extract_size(size, source_volume, snapshot):提取并验证卷的大小；

def _check_image_metadata(self, context, image_id, size):检测镜像的存在性，并验证镜像的元数据中镜像大小的属性信息；

def _check_metadata_properties(metadata=None):检测卷的元数据属性是有效的；

def _extract_availability_zone(self, availability_zone, snapshot,source_volume):提取并返回一个经过验证的可用的 zone；

def _get_encryption_key_id(self, key_manager, context, volume_type_id,snapshot, source_volume, backup_source_volume):获取加密密钥信息的 id 值；

def _get_volume_type_id(self, volume_type, source_volume, snapshot,backup_source_volume):获取卷类型信息的 id 值；

def __call__(self, context, size, snapshot, image_id, source_volume,availability_zone, volume_type, metadata, key_manager, backup_source_volume):这个 task 的主要任务是提取和验证输入的值，这些输入的值将形成一个潜在的卷的请求信息；并且实现根据一组条件对这些输入值进行验证，并将这些输入值转换成有效的集合；并返回这些经过验证和转换的输入值，这些输入值将会应用于其他 task 中。

class EntryCreateTask(base.CinderTask):在数据库中为给定的卷建立条目；逆转操作：从数据库中删除 volume_id 建立的条目；

def __call__(self, context, **kwargs):为给定的输入在数据库中建立数据条目，并返回详细信息；从 kwargs 中获取卷的相关属性值，并根据卷的相关属性值在数据库中实现新卷的建立；

def revert(self, context, result, cause):删除指定的卷在数据库中的数据条目信息，实现逆转回滚操作；

class QuotaReserveTask(base.CinderTask):根据给定的大小值和给定的卷类型信息实现保存单一的卷；

def __call__(self, context, size, volume_type_id):根据给定的大小值和给定的卷类型信息实现保存单一的卷；

def revert(self, context, result, cause):回调配额预留资源；

class QuotaCommitTask(base.CinderTask):提交资源配额的预留信息到数据库中；

def __call__(self, context, reservations, volume_properties):提交资源配额的预留信息到数据库中；

def revert(self, context, result, cause):实现逆转回滚操作；

class VolumeCastTask(base.CinderTask):远程调用实现卷的建立操作；

def _cast_create_volume(self, context, request_spec, filter_properties):远程调用建立卷的方法，实现卷的建立操作；

def __call__(self, context, **kwargs):远程调用实现卷的建立操作；

class OnFailureChangeStatusTask(base.CinderTask):这个 task 实现了当出现错误时，设置指定 id 的卷的状态为 ERROR；

class OnFailureRescheduleTask(base.CinderTask):触发一个发送进行重新调度的请求，当进行 task 恢复回滚操作的时候；

def __call__(self, context, *args, **kwargs):触发一个发送进行重新调度的请求，当进行 task 恢复回滚操作的时候；

def _reschedule(self, context, cause, request_spec, filter_properties,snapshot_id, image_id, volume_id, **kwargs):实现重新调度的操作，并实现重新卷的建立操作；

def _pre_reschedule(self, context, volume_id):实现重新调度操作前进行的一些操作；

def revert(self, context, result, cause):实现任务 task 的回滚操作；

class NotifySchedulerFailureTask(base.CinderTask):当任务出现错误后用于通知任务错误的 task；

class ExtractSchedulerSpecTask(base.CinderTask):实现了从输入的参数中提取对象的规范信息的操作；

class ExtractVolumeSpecTask(base.CinderTask):提取一个用于建立卷的通用结构规范；

def __call__(self, context, volume_id, **kwargs):提取一个用于建立卷的通用结构规范；即获取 specs 和 volume_ref 的数据信息；

class NotifyVolumeActionTask(base.CinderTask):执行关于给定卷的相关通知操作；获取指定卷的使用率信息，并进行通知操作；

class CreateVolumeFromSpecTask(base.CinderTask):根据所提供的规范要求实现卷的建立操作；

def _handle_bootable_volume_glance_meta(self, context, volume_id,**kwargs):根据具体情况实现对指定卷的 glance 元数据进行更新操作；调用者应该提供 snapshot_id、source_volid 和 image_id 三个参数之一，如果提供的是 image_id，则还应该提供 image_meta，否则则被视为空的字典；

def _create_from_snapshot(self, context, volume_ref, snapshot_id,**kwargs):实现从快照建立卷的操作，并根据具体情况实现对指定卷的 glance 元数据进行更新操作；

def _enable_bootable_flag(self, context, volume_id):确认 bootable 标志的值；

def _create_from_source_volume(self, context, volume_ref, source_volid,**kwargs):实现从源卷建立（实际上就是直接拷贝）卷的操作；

def _copy_image_to_volume(self, context, volume_ref, image_id,image_location, image_service):下载 glance 镜像数据到指定的卷；

def _capture_volume_image_metadata(self, context, volume_id, image_id,image_meta):根据给定的 image_meta 为指定的卷更新 Glance 元数据；

def _create_from_image(self, context, volume_ref, image_location,image_id, image_meta, image_service, **kwargs):从镜像实现卷的建立；

def _create_raw_volume(self, context, volume_ref, **kwargs):实现 raw 格式卷的建立；

def __call__(self, context, volume_ref, volume_spec):根据所提供的规范要求实现卷的建立操作；

class CreateVolumeOnFinishTask(NotifyVolumeActionTask):当成功的建立卷之后，完成卷建立之后的通知操作；

def get_api_flow(scheduler_rpcapi,volume_rpcapi, db, image_service, az_check_functor, create_what):构建并返回用于建立卷的 flow；flow 将会做如下的事情：1. 为相关的 task 注入 keys 和 values；2. 提取并验证输入的 keys 和 values；3. 保留配额数据（如果有错误，则恢复配额数据）；4. 在数据库中建立条目；5. 提交资源配额的预留信息到数据库中；6. 远程调用实现卷的建立操作；

def get_scheduler_flow(db, driver,request_spec=None, filter_properties=None, volume_id=None, snapshot_id=None,image_id=None):构建并返回用于通过远程调度建立卷的 flow；flow 将会做以下的事情：1. 为相关的 task 注入 keys 和 values；2. 实现了从输入的参数中提取调度器的规范信息的操作；3. 对于出错的 task 进行处理，发送错误通知，记录错误信息等；4. 远程调用实现在主机上建立卷；

def get_manager_flow(db, driver,scheduler_rpcapi, host, volume_id, request_spec=None, filter_properties=None,allow_reschedule=True, snapshot_id=None, image_id=None, source_volid=None,reschedule_context=None):构建并返回用于通过管理器建立卷的 flow；flow 将会做以下的事情：1. 首先要确定我们是否允许进行重新调度，因为这影响了我们如何对出现错误的情况进行处理；2. 为相关的 task 注入 keys 和 values；3. 对于出错的 task 进行处理，发送错误通知，记录错误信息等；4. 实现了从输入的参数中提取建立卷的规范信息的操作；5. 通知已经开始进行卷的建立操作；6. 根据所获取的建立卷的规范信息实现卷的建立操作；7. 当成功的建立卷之后，完成卷建立之后的通知操作；

/cinder/volume/drivers/

/cinder/volume/drivers/emc----emc 卷存储驱动；

/cinder/volume/drivers/hds----hus 卷存储驱动；

/cinder/volume/drivers/huawei----huawei 卷存储驱动；

/cinder/volume/drivers/netapp----NetApp 卷存储驱动；

/cinder/volume/drivers/nexenta----Nexenta 卷存储驱动；

/cinder/volume/drivers/san----San 卷存储驱动；

/cinder/volume/drivers/vmware----VMware 卷存储驱动；

/cinder/volume/drivers/windows----Windows 卷存储驱动；

/cinder/volume/drivers/xenapi----XenApi 卷存储驱动；

/cinder/volume/drivers/coraid.py----Coraid 卷存储驱动；

/cinder/volume/drivers/eqlx.py----DELL 卷存储驱动；

/cinder/volume/drivers/gpfs.py----GPFS 卷存储驱动；

/cinder/volume/drivers/nfs.py----NFS 卷存储驱动；

/cinder/volume/drivers/rbd.py----RADOS 块设备驱动；

/cinder/volume/drivers/scality.py----ScalitySOFS 卷驱动；

/cinder/volume/drivers/solidfire.py----SolidFire 卷驱动；

/cinder/volume/drivers/storwize_svc.py----IBMStorwize 和 SVC 卷驱动；

/cinder/volume/drivers/xiv_ds8k.py----IBMXIV 和 DS8K 存储系统卷驱动；

/cinder/volume/drivers/zadara.py----VPSA 卷驱动；

/cinder/volume/drivers/glusterfs.py----Glusterfs 卷存储驱动；

class GlusterfsDriver(nfs.RemoteFsDriver):Glusterfs 文件系统驱动；

def do_setup(self, context):启动的时候进行卷驱动的初始化操作；

def _local_volume_dir(self, volume):根据卷的属性确定本地卷的直接路径；

def _local_path_volume(self, volume):根据卷的属性确定本地卷的全路径；

def _local_path_volume_info(self, volume):根据卷的属性确定本地卷的全路径信息；

def _qemu_img_info(self, path):从路径中获取 qemu-img 信息；

def get_active_image_from_info(self, volume):从指定的卷中获取状态为活跃的镜像快照的文件名；

def create_cloned_volume(self, volume, src_vref):建立指定卷的拷贝；

def create_volume(self, volume):实现建立指定的卷操作；

def create_volume_from_snapshot(self, volume, snapshot):实现从快照建立卷的操作（要求用于建立卷的快照的状态是活跃的）；

def _copy_volume_from_snapshot(self, snapshot, volume, volume_size):实现从快照拷贝数据到目标卷的操作；

def delete_volume(self, volume):实现删除一个逻辑卷的操作；

def create_snapshot(self, snapshot):实现建立快照的操作；

def _create_qcow2_snap_file(self, snapshot, backing_filename,new_snap_path):建立 QCOW2 格式的快照文件，并备份于指定的用于备份的文件中；

def _create_snapshot(self, snapshot, path_to_disk, snap_id):实现建立快照的操作（离线状态下）；

def _read_file(self, filename):实现都取指定文件内容信息；

def _read_info_file(self, info_path, empty_if_missing=False):获取包含快照信息的字典；

def _write_info_file(self, info_path, snap_info):将 snap_info 中的数据写入文件路径 info_path 中；

def delete_snapshot(self, snapshot):实现删除一个快照的操作；

def _delete_snapshot_online(self, context, snapshot, info):实现在线删除快照文件的操作；

def ensure_export(self, ctx, volume):为指定的逻辑卷重建一个导出；

def create_export(self, ctx, volume):实现导出卷的操作；

def remove_export(self, ctx, volume):实现删除指定逻辑卷的导出；

def initialize_connection(self, volume, connector):允许连接到连接器，并返回连接信息；

def terminate_connection(self, volume, connector, **kwargs):断开到连接器的链接；

def copy_volume_to_image(self, context, volume, image_service,image_meta):拷贝卷到指定的镜像 image_service；

def extend_volume(self, volume, size_gb):根据给定镜像大小值，实现为给定卷路径的卷的扩展操作；

def _do_create_volume(self, volume):实现在给定的 GlusterFS 共享文件系统上建立卷的操作；

def _ensure_shares_mounted(self):挂载所有配置 GlusterFS 共享文件，并存储所有的挂载点信息；

def _ensure_share_mounted(self, glusterfs_share):实现挂载 GlusterFS 共享文件系统；

def _find_share(self, volume_size_for):根据给定的卷大小在多个可用的 GlusterFS 共享文件系统中选取合适的 GlusterFS 共享文件系统；目前的实现是根据所拥有最大的 capacity 数据来进行确定；

def _get_hash_str(self, base_str):获取 hash 字符串；

def _get_mount_point_for_share(self, glusterfs_share): 为共享文件获取挂载点（例子：172.18.194.100:/var/glusterfs）；

def _get_available_capacity(self, glusterfs_share):计算 GlusterFS 共享文件的可用空间信息；

def _get_capacity_info(self, glusterfs_share):获取 GlusterFS 共享文件的 capacity 信息；

def _mount_glusterfs(self, glusterfs_share, mount_path, ensure=False):实现挂载 GlusterFS 共享磁盘到指定的路径 mount_path 上；

/cinder/volume/drivers/lvm.py----LVM 卷存储驱动；

class LVMVolumeDriver(driver.VolumeDriver):执行与卷相关的命令行；

def _delete_volume(self, volume, is_snapshot=False):实现删除一个逻辑卷操作；

def _create_volume(self, name, size, lvm_type, mirror_count, vg=None):在对象 VG 上建一个逻辑卷；

def create_volume(self, volume):实现建立逻辑卷；

def create_volume_from_snapshot(self, volume, snapshot):实现从给定的快照建立卷的操作；

def delete_volume(self, volume):实现删除一个逻辑卷的操作；

def clear_volume(self, volume, is_snapshot=False):清除旧的卷数据以防止用户之间的数据泄露；

def create_snapshot(self, snapshot):实现为逻辑卷建立快照；

def delete_snapshot(self, snapshot):实现删除一个逻辑卷快照操作；

def local_path(self, volume, vg=None):获取卷的本地路径信息；

def copy_image_to_volume(self, context, volume, image_service,image_id):实现从 image_service 中获取镜像数据，并写入到卷中；

def copy_volume_to_image(self, context, volume, image_service,image_meta):实现拷贝卷的数据到指定的镜像；

def create_cloned_volume(self, volume, src_vref):实现建立一个指定卷的拷贝操作；

def backup_volume(self, context, backup, backup_service):实现从一个已存在的卷建一新备份操作；

def restore_backup(self, context, backup, volume, backup_service):还原一个现有的备份到一个新的或者是已存在的卷；

def get_volume_stats(self, refresh=False):获取卷的状态；如果 refresh 的值为 True，则进行先更新卷的状态操作；

def _update_volume_stats(self):从卷组检索获取统计数据信息；

def extend_volume(self, volume, new_size):为存在的卷实现大小的扩展；

class LVMISCSIDriver(LVMVolumeDriver, driver.ISCSIDriver):执行与 ISCSI 卷相关的命令行；

def _create_tgtadm_target(self, iscsi_name, iscsi_target, volume_path,chap_auth, lun=0, check_exit_code=False, old_name=None):实现建立 tgtadm 目标操作；

def ensure_export(self, context, volume):实现为逻辑卷同步重建导出的操作；

def _ensure_iscsi_targets(self, context, host):确认 target 在数据存储系统中已经建立；

def create_export(self, context, volume):实现建立逻辑卷的导出操作；

def _create_export(self, context, volume, vg=None):实现建立一个逻辑卷的导出操作；

def remove_export(self, context, volume):实现删除一个逻辑卷的导出操作；

def migrate_volume(self, ctxt, volume, host, thin=False,mirror_count=0):卷的迁移操作；如果卷的迁移的目标节点和源节点在相同的服务器上，则优化迁移操作；

class LVMISERDriver(LVMISCSIDriver,driver.ISERDriver):执行与 ISER 卷相关的命令行；

def ensure_export(self, context, volume):实现为逻辑卷同步重建导出的操作；

def _ensure_iser_targets(self, context, host):确认 target ids 已经在数据存储中建立；

def create_export(self, context, volume):实现建立逻辑卷的导出操作；

def remove_export(self, context, volume):实现删除一个逻辑卷的导出操作；

/cinder/volume/drivers/sheepdog.py----sheepdog 卷驱动；

class SheepdogDriver(driver.VolumeDriver):执行和 Sheepdog 存储卷相关的命令；

def check_for_setup_error(self):如果必要条件不符合，则返回错误信息；

def create_volume(self, volume):实现建立一个 sheepdog 卷的操作；

def create_volume_from_snapshot(self, volume, snapshot):实现从快照建立一个 sheepdog 卷的操作；

def delete_volume(self, volume):实现删除一个逻辑卷的操作；

def _ensure_dir_exists(self, tmp_dir):检测 tmp_dir 是否存在，如不存在的话，建立其指定的文件夹；

def _resize(self, volume, size=None):实现调整卷的大小操作；

def _delete(self, volume):实现删除一个逻辑卷的操作；

def copy_image_to_volume(self, context, volume, image_service,image_id):拷贝镜像到卷的操作；

def create_snapshot(self, snapshot):实现建立一个 sheepdog 快照的操作；

def delete_snapshot(self, snapshot):实现删除一个 sheepdog 快照的操作；

def ensure_export(self, context, volume):实现重建并导出逻辑卷的操作；

def create_export(self, context, volume):实现导出卷的操作；

def remove_export(self, context, volume):实现删除一个逻辑卷的导出的操作；

def initialize_connection(self, volume, connector):实现初始化到连接器的连接操作；

def terminate_connection(self, volume, connector, **kwargs):实现中止到连接器的连接操作；

def _update_volume_stats(self):实现更新卷的统计数据信息的操作；

def get_volume_stats(self, refresh=False):实现获取卷的统计数据信息的操作；

def extend_volume(self, volume, new_size):扩展一个已存在的卷的大小；

def backup_volume(self, context, backup, backup_service):为已存在的卷建立一个新的备份；

def restore_backup(self, context, backup, volume, backup_service):还原一个现有的备份为新的卷或者已存在的卷；

10 db（/cinder/db/）

/cinder/db/api.py：定义 DB 的若干接口入口方法；

/cinder/db/base.py：需要模块化的数据库接口的基类；

/cinder/db/migration.py：主要定义和实现了数据库版本操作的两个方法；

/cinder/db/sqlalchemy/api.py：针对数据库相关的 api 方法（/cinder/db/api.py）的具体实现的 api 方法；

/cinder/db/sqlalchemy/migration.py：

/cinder/db/sqlalchemy/models.py：定义和描述了 cinder 相关数据表的结构；

class CinderBase-Cinder Model 的基类；

class Service(BASE, CinderBase):表示一个主机上正在运行的服务相关数据库中的信息；

class CinderNode(BASE, CinderBase):表示一个主机上正在运行的 cinder 服务相关数据库中的信息；

class Volume(BASE, CinderBase):表示一个能够附加到虚拟机的块存储设备的相关数据库中的信息；

class VolumeMetadata(BASE, CinderBase):表示一个卷的元数据的键值对；

class VolumeAdminMetadata(BASE, CinderBase):表示一个管理员所有的卷的元数据的键值对；

class VolumeTypes(BASE, CinderBase):表示卷可能支持的所有卷类型的数据库相关信息；

class VolumeTypeExtraSpecs(BASE, CinderBase):表示卷类型的额外功能（规范）的键值对；

class QualityOfServiceSpecs(BASE, CinderBase):表示 QoS 功能（规范）的键值对；

class VolumeGlanceMetadata(BASE, CinderBase):表示可引导卷的 glance 元数据；

class Quota(BASE, CinderBase):表示一个单一的对象的配额信息；

class QuotaClass(BASE, CinderBase):表示一个配额类（quota class）的配额信息；

class QuotaUsage(BASE, CinderBase):表示指定资源当前使用率的数据库信息；

class Reservation(BASE, CinderBase):表示资源配额恢复的相关数据库信息；

class Snapshot(BASE, CinderBase):表示一个能够附加到虚拟机块存储设备（也就是快照）的数据库信息；

class SnapshotMetadata(BASE, CinderBase):表示一个快照的元数据的键值对数据库相关信息；

class IscsiTarget(BASE, CinderBase):表示一个指定主机的 iscsi target 的数据库信息；

class Backup(BASE, CinderBase):表示针对 Swift 后端的卷的备份；

class Encryption(BASE, CinderBase):表示一个卷类型的加密请求的相关数据库信息；

class Transfer(BASE, CinderBase):表示一个卷所有权转换请求的相关数据库信息；