Cinder卷迁移代码分析

migrate代码分析：

Cinder-API:

cinder.volume.api.API

接口：def migrate\_volume(self, context, volume, host, cluster\_name, force\_copy,lock\_volume)

主要工作内容：

1.检查传入的host信息是否合法，即backend进程是否实际存在；

2.检查卷的状态是否合法，要求如下：

status : ('available', 'in-use')

migration\_status : (None, 'deleting', 'error', 'success')

replication\_status : (None, 'disabled')

consistencygroup\_id : (None, '')

group\_id : (None, '')

3.同时迁移时要求卷不能存在快照

4.更新卷的migration\_status和status（如果配置了lock\_volume更新status为 maintenance）

调用cinder-scheduler的迁移接口：

volume\_type = {}  
**if** volume.volume\_type\_id:  
 volume\_type = volume\_types.get\_volume\_type(context.elevated(),volume.volume\_type\_id)  
request\_spec = {'volume\_properties': volume,  
 'volume\_type': volume\_type,  
 'volume\_id': volume.id}  
self.scheduler\_rpcapi.migrate\_volume(context, volume, cluster\_name **or** host, force\_copy,  
 request\_spec)

Cinder-Scheduler：

cinder. scheduler.manager. SchedulerManager:

接口：

def migrate\_volume(self, context, volume, backend, force\_copy, request\_spec, filter\_properties)

主要工作内容：

1. 根据volume的信息判断过滤所有合适的pool，判断目标卷是否在上一步过滤出的卷中，如果不在那么迁移失败。

注意，到在migrate\_volume方法中调用的接口是backend\_passes\_filters，这个接口和创建volume时使用的接口是同一个接口，因此c-sch在迁移时会对pool按照配置的过滤器已进行一系列过滤。因此对于一些跨Backend迁移的情形，可能会导致由于volume\_type中的volume\_backend\_name不同导致无法迁移的情况。（比如想要迁移的设备是A，但是卷的volume\_type中配置volume\_backend\_name=b。那么backend\_passes\_filters会过滤出所有volume\_backend\_name=b的pool，但是A并不在过滤出的结果中，迁移无法进行！）

**def migrate\_volume**(self, context, volume, backend, force\_copy,  
 request\_spec, filter\_properties):  
 *"""Ensure that the backend exists and can accept the volume."""* self.\_wait\_for\_scheduler()  
 **try**:  
 tgt\_backend = self.driver.backend\_passes\_filters(context, backend,  
 request\_spec,  
 filter\_properties)  
 **except** exception.NoValidBackend **as** ex:  
 \_migrate\_volume\_set\_error(self, context, ex, request\_spec)  
 **except** Exception **as** ex:  
 **with** excutils.save\_and\_reraise\_exception():  
 \_migrate\_volume\_set\_error(self, context, ex, request\_spec)  
 **else**:  
 volume\_rpcapi.VolumeAPI().migrate\_volume(context, volume,  
 tgt\_backend,  
 force\_copy)

1. 调用cinder-volume的迁移接口。

Cinder-Volume:

cinder. volume.manager. VolumeManager:

接口：***def migrate\_volume(self, ctxt, volume, host, force\_host\_copy=False, new\_type\_id=None)***

主要功能：

1.更新卷的migration\_status为migrating

2.尝试在storage内部进行卷迁移

当force\_host\_copy值为false时c-vol会优先进行storage内部的卷迁移（new\_type\_id这个值始终为None，这个入参实际上没有被用到，一致使用的是默认值None）。

self.driver.migrate\_volume(ctxt,volume,host)调用驱动的迁移代码，返回值moved表示是否已经完成了卷迁移，model\_update表示迁移过程中需要更新到DB的信息（比如wwn信息变动等）。

**if not** force\_host\_copy **and** new\_type\_id **is** None:  
 **try**:  
 LOG.debug("Issue driver.migrate\_volume.", resource=volume)  
 moved, model\_update = self.driver.migrate\_volume(ctxt,  
 volume,  
 host)  
 **if** moved:  
 updates = {'host': host['host'],  
 'cluster\_name': host.get('cluster\_name'),  
 'migration\_status': 'success',  
 'previous\_status': volume.status}  
 **if** status\_update:  
 updates.update(status\_update)  
 **if** model\_update:  
 updates.update(model\_update)  
 volume.update(updates)  
 volume.save()  
 **except** Exception:  
 **with** excutils.save\_and\_reraise\_exception():  
 updates = {'migration\_status': 'error'}  
 **if** status\_update:  
 updates.update(status\_update)  
 volume.update(updates)  
 volume.save()

3.当storage内部迁移失败，moved的返回值为false时，c-vol调用接口在节点上强制执行迁移。强制迁移接口：

self.\_migrate\_volume\_generic(ctxt, volume, host, new\_type\_id)

cinder. volume.manager. VolumeManager:

接口：def \_migrate\_volume\_generic(self, ctxt, volume, backend, new\_type\_id

主要功能：

1. 在targethost上创建一个空白卷
2. 判断Volume是否attach（在Volume\_attachment表中是否有记录），如果没有c-vol开始对卷进行冷迁移，否者进行热迁移。

从下面的代码可以发现，冷迁时c\_vol调用\_copy\_volume\_data接口将源卷中的内容拷贝到目标卷，拷贝完成后migrate\_volume\_completion接口

# Copy the source volume to the destination volume  
**try**:  
 attachments = volume.volume\_attachment  
 **if not** attachments:  
 # Pre- and post-copy driver-specific actions  
 self.driver.before\_volume\_copy(ctxt, volume, new\_volume,  
 remote='dest')  
 self.\_copy\_volume\_data(ctxt, volume, new\_volume, remote='dest')  
 self.driver.after\_volume\_copy(ctxt, volume, new\_volume,  
 remote='dest')  
 # The above call is synchronous so we complete the migration  
 self.migrate\_volume\_completion(ctxt, volume, new\_volume,  
 error=False)  
 **else**:  
 nova\_api = compute.API()  
 # This is an async call to Nova, which will call the completion  
 # when it's done  
 **for** attachment **in** attachments:  
 instance\_uuid = attachment['instance\_uuid']  
 nova\_api.update\_server\_volume(ctxt, instance\_uuid,  
 volume.id,  
 new\_volume.id)

cinder. volume.manager. VolumeManager:

def migrate\_volume\_completion (self, ctxt, volume, new\_volume, error=False)

该接口的作用是：当迁移copy完成以后，对卷的状态信息进行更新，以及对volume进行dettach/attach状态更新。

进行冷迁移时：调用者是c-vol本身

1. 调用update\_migrated\_volume接口将new\_volume的id和provider\_location更新到volume的\_name\_id和provider\_location中（这里如何更新优先通过驱动的update\_migrated\_volume接口，如果驱动没有实现这个接口，那么按上述方案来）。
2. 调用finish\_volume\_migration接口，将volume中的信息更新new\_volume中，准备删除new\_volume
3. 通过上述两步实际上new\_volume和volume对应的LUN完全互换了，因此删除时通过new\_volume记录删除旧的LUN

在冷迁移时上述三个步骤完成后，卷的迁移就完成了。

进行热迁移时：调用者是n-cpu（因为对LUN进行copy的工作时n\_cpu做的）

1. 将volume的所有attachments状态更新为dettach
2. 调用update\_migrated\_volume接口
3. 调用finish\_volume\_migration接口
4. 将volume的所有attachments状态更新为attach
5. 删除new\_volume记录对应的LUN（即旧的LUN）

需要注意的是这里dettach/attach动作是nova完成的，cinder只是负责更新状态。

磁盘热迁移时nova代码逻辑分析：

热迁移时主要代码逻辑在nova-compute中。迁移cinder一旦检查到卷时in-use状态，会通知nova进行卷拷贝（cinder会事先提供好一个用来拷贝的空盘）。nova完成copy动作后调用cinder的migrate\_volume\_completion接口，通知cinder删除源卷。

Nova\_API

接口方法：

nova.compute.api.API :

def swap\_volume(self, context, instance, old\_volume, new\_volume)

context：request上下文信息；

instance：虚机实例信息

old\_volume：待迁移的卷信息

new\_volume：迁移的目标卷，此时是一个空盘

swap\_volume接口的任务是将old\_volume中的数据全部写入new\_volume中，并且完成连接的切换。

关键代码：

# 确保old\_volume当前时attach状态  
self.volume\_api.check\_detach(context, old\_volume)  
# 确保new\_volume当前时dettach状态  
self.volume\_api.check\_attach(context, new\_volume, instance=instance)  
# 通知cinder 将old\_volume状态设定为detaching，注意这里只是改了Cinder那边的状态  
self.volume\_api.begin\_detaching(context, old\_volume['id'])  
# 通知cinder 将new\_volume状态设定为attaching，注意这里只是改了Cinder那边的状态  
self.volume\_api.reserve\_volume(context, new\_volume['id'])  
**try**:  
 #调用nova\_cpu进行卷交换  
 self.compute\_rpcapi.swap\_volume(  
 context, instance=instance,  
 old\_volume\_id=old\_volume['id'],  
 new\_volume\_id=new\_volume['id'])  
**except** Exception:  
 #出现失败时回滚old\_volume和new\_volume的状态  
 **with** excutils.save\_and\_reraise\_exception():  
 self.volume\_api.roll\_detaching(context, old\_volume['id'])  
 self.volume\_api.unreserve\_volume(context, new\_volume['id']

Nova\_Compute

接口方法：

nova.compute.manager.ComputeManager:

def swap\_volume(self, context, old\_volume\_id, new\_volume\_id, instance)

1. 调用\_swap\_volume(context, instance, bdm, connector,old\_volume\_id, new\_volume\_id, resize\_to)接口，该接口会调用libvirt进行磁盘拷贝于磁盘交换。
2. 更新数据库中的磁盘映射信息（主要是block\_device\_mapping表）

def \_swap\_volume(context, instance, bdm, connector,old\_volume\_id, new\_volume\_id, resize\_to)

1. 通过cinder获取old\_volume和new\_volume的连接信息
2. 调用驱动的swap\_volume接口
3. 切断虚机关于旧LUN的连接
4. 调用cinder的migrate\_volume\_completion接口通知cinder迁移copy动作已经完成，可以进行下一步操作。

Libvirt层的操作（nova.virt.libvirt.driver.LibvirtDriver）：

接口：

def swap\_volume(self, old\_connection\_info, new\_connection\_info, instance, mountpoint, resize\_to)

1. 建立n-cpu节点和new\_volume（new\_connection\_info）的连接；
2. 调用\_swap\_volume接口，该接口是nova真正拷贝的接口调用了以下方法：

nova.virt.libvirt.guest.BlockDevice

* 1. abort\_job()

同步终止关于硬盘的所有读写操作

* 1. rebase(new\_path, copy=True, reuse\_ext=True)

在路径new\_path（即新硬盘）为磁盘创建一个备份，同时reuse\_ext=True指定了在拷贝过程中还可以写硬盘，但是此时不是写在块设备中，而是写在nova本地的一个文件中。

* 1. dev.abort\_job(pivot=True)

停止磁盘的读写，并且将b)过程中的读写内容合并