Cinder 磁盘拷贝流程分析

1.场景

在migration过程中需要对磁盘中的文件进行拷贝

```
cinder.volume.manager.VolumeManager
def _copy_volume_data(self, ctxt, src_vol, dest_vol, remote=None)
```

上述方法会同时挂载src_vol和dest_vol对应的磁盘，进行数据拷贝。

attach步骤：

1.获取连接信息：

```
from cinder import utils
properties = utils.brick_get_connector_properties()
```

cinder.utils中的brick_get_connector_properties方法调用的是os-brick中的get_connector_properties方法，这点是和Nova进行attach是一样的。

2.建立连接

```
 Cinder.volume.manager.VolumeManager对应的接口：



    def _attach_volume(self, ctxt, volume, properties,
remote=False,
                       attach_encryptor=False):
    """
    _attach_volume是cinder-volume挂载磁盘时候的接口函数
    :param ctxt:
    :param volume: 要挂载的磁盘
    :param properties:本机的连接信息
    :param remote:当前磁盘是否在本地，及这个磁盘是否属于当前
VolumeManager的backend
    :param attach_encryptor: 磁盘是否编码加密
    :return:返回连接信息
    """
```

整个过程如下：

1. Cinder.volume.manager.VolumeManager的initialize_connection是获取target的接口，该方法会调用驱动的create_export和initialize_connection返回target信息。
2. 连接设备接口Cinder.volume.manager.VolumeManager的_connect_device：
   获取conn["driver_volume_type"]，初始化os-brick的驱动
   connector（cinder.utils.brick_get_connector）
   通过connector，conn["data"]建立连接，返回connection信息！！！

在创建连接的过程中，volume卷的状态不会改变，不会生成attachments记录，这一点需要注意。
同时调用initialize_connection方法时，需要和volume的具体driver进行通讯，因此在多节点环境下需要使用
rpc接口进行调用：

```
         rpcapi = volume_rpcapi.VolumeAPI()
        try:
            conn = rpcapi.initialize_connection(ctxt, volume,
properties)
        except Exception:
            with excutils.save_and_reraise_exception():
                LOG.error(_LE("Failed to attach volume %
(vol)s."),
                          {'vol': volume['id']})
                self.db.volume_update(ctxt, volume['id'],
                                      {'status': status})
```

Copy步骤：

接口：

```
#cinder.volume.utils

def copy_volume(src, dest, size_in_m, blocksize, sync=False,
            execute=utils.execute, ionice=None, throttle=None,
            sparse=False):
"""
将src中的内容拷贝到dest，这里src和dest是两个字符串，被copy_volume假象成文件系统中的两个文件，
拷贝时使用的命令时dd
:param src: 源卷
:param dest: 目标卷
:param size_in_m: 源卷的大小（MB）
:param blocksize: 拷贝时的块大小，通过CONF.volume_dd_blocksize可以指定默认1M
:param sync:把每个输入块进行填充，不足部分用空(NUL)字符补齐
:param execute:root_wrap
:param ionice:IO任务的优先级，可以参考命令ionice
:param throttle:带宽限制，默认时不限制带宽
:param sparse:将dest转换稀疏格式的，这个取决于驱动
:return:
"""

if (isinstance(src, six.string_types) and
        isinstance(dest, six.string_types)):
    if not throttle:
        # throttle的subcommand的prefix可以限制copy速度？为空表示不限制？
        # 关系到的配置项volume_copy_bps_limit
        throttle = throttling.Throttle.get_default()
    with throttle.subcommand(src, dest) as throttle_cmd:

        # throttle_cmd['prefix'] = []
        # sparse=True (netapp时)
        # sync=False
        # inice=None
        _copy_volume_with_path(throttle_cmd['prefix'], src, dest,
```

```
                                    size_in_m, blocksize, sync=sync,
                                    execute=execute, ionice=ionice,
                                    sparse=sparse)
else:
    _copy_volume_with_file(src, dest, size_in_m)
```

_copy_volume_with_path使用的是"dd"命令进行拷贝，有以下能力：

1.可以使用ionice调节IO进程的优先级；

2.可以通过cinder.volume.throttling.Throttle来限制传输带宽？？

3.使用direct语义写盘，会事先检查是否支持direct
dd count=0 if= of= oflag=direct

4.可以将dest转换成稀疏格式的，这个取决于驱动

```
dd if=<PATH_1> of=<PATH_2> count=10737418240 bs=1M
iflag=count_bytes,direct oflag=direct conv=sparse
```

5.同等条件下_copy_volume_with_file只有_copy_volume_with_path的三分之一，原因是_copy_volume_with_file是用Python的文件操作接口进行拷贝。

6.在172.24.2.216上的速度测试：

```
#CONF.volume_dd_blocksize=1M

size 10240.00 MB, duration 154.07 sec
size 30720.00 MB, duration 466.48 sec

#CONF.volume_dd_blocksize=64M

size 10240.00 MB, duration 157.72 sec
```

detach步骤

接口：

```
#cinder.volume.manager.VolumeManager
 def _detach_volume(self, ctxt, attach_info, volume, properties,
                    force=False, remote=False,
                    attach_encryptor=False):
    """
    断开磁盘连接
    :param ctxt:
    :param attach_info:连接信息 ， 主要包含attach_info['connector']
信息,该变量是对应的os-brick驱动
    :param volume:
    :param properties: 本机信息
    :param force: 是否强制断开连接
    :param remote: volume是否属于当前VolumeManager管理
    :param attach_encryptor: volume是否加密
    :return:
    """
    connector = attach_info['connector']
    if attach_encryptor and (
            volume_types.is_encrypted(ctxt,
                                      volume.volume_type_id)):
        encryption = self.db.volume_encryption_metadata_get(
            ctxt.elevated(), volume.id)
        if encryption:
            utils.brick_detach_volume_encryptor(attach_info,
encryption)
    connector.disconnect_volume(attach_info['conn']['data'],
                                attach_info['device'])

    if remote:
        rpcapi = volume_rpcapi.VolumeAPI()
        rpcapi.terminate_connection(ctxt, volume, properties,
force=force)
        rpcapi.remove_export(ctxt, volume)
    else:
        try:
            self.terminate_connection(ctxt, volume['id'],
```

```
properties,
                                       force=force)
            self.remove_export(ctxt, volume['id'])
        except Exception as err:
            with excutils.save_and_reraise_exception():
                LOG.error(_LE('Unable to terminate volume
connection: '
                              '%(err)s.') % {'err': err})
```

在Cinder执行detach操作时，主要有以下两个步骤：

1.通过attach_info['connector']与os-brick交互，通知磁盘使用者清理连接信息，如：iscsi登出，清理iscsi驱动设备等。

```
connector.disconnect_volume(attach_info['conn']
['data'],attach_info['device'])
```

2.通知target断开连接，如iscsi需要target清理LUN的映射信息。该步骤实际上并非所有存储协议都需要，有些不需要。该步骤是VolumeManager的驱动和Storage后端直接交互。

```
def terminate_connection(self, context, volume_id, connector,
force=False):
    """Cleanup connection from host represented by connector.

    The format of connector is the same as for
initialize_connection.
    """

    utils.require_driver_initialized(self.driver)

    volume_ref = self.db.volume_get(context, volume_id)
    try:
        # 通知target清理连接
        self.driver.terminate_connection(volume_ref, connector,
                                         force=force)
    except Exception as err:
        err_msg = (_('Terminate volume connection failed: %
(err)s')
                   % {'err': six.text_type(err)})
        LOG.exception(err_msg, resource=volume_ref)
        raise exception.VolumeBackendAPIException(data=err_msg)
    LOG.info(_LI("Terminate volume connection completed
successfully."),
             resource=volume_ref)

def remove_export(self, context, volume_id):
    """Removes an export for a volume."""
    utils.require_driver_initialized(self.driver)
    volume_ref = self.db.volume_get(context, volume_id)
    try:
        self.driver.remove_export(context, volume_ref)
    except Exception:
        msg = _("Remove volume export failed.")
        LOG.exception(msg, resource=volume_ref)
        raise exception.VolumeBackendAPIException(data=msg)
```