**Cinder磁盘快照功能分析**

# 创建快照

## API接口

http://<addr\_ip>:8776/v2/<project\_id>/snapshots

body: {

"snapshot": {

"description": null,

"metadata": {

"key1":"value1",

"key2":"value2",

......

},

"force": "True",

"name": "display\_name",

"volume\_id": "source\_volid"

}

}

# 参数说明：

# metadata： 快照的源数据信息

# description： 快照的描述信息

# force： 是否创建强制快照，为False时无法对in-use状态的卷进行快照，为True时忽略卷的Attachments信息进行快照。

# name： 快照的名称

# volume\_id： 快照的源卷ID

## 代码逻辑分析

1. Cinder-API

*Snapshot对应 wsgi的类为cinder.api.v2.snapshots.SnapshotsController。*

*分析SnapshotsController方法可以看到，创建Snapshot时，根据force的值，存在两个逻积处理的分支，分别对不同状态的卷进行快照。*

@wsgi.response(202)

def create(self, req, body):

"""Create 检查body中的False参数的值，根据force的值选择不同的逻辑分支"""

......

# 根据force的值调用不同的逻辑处理接口

if force:

new\_snapshot = self.volume\_api.create\_snapshot\_force(

context,

volume,

snapshot.get('display\_name'),

snapshot.get('description'),

\*\*kwargs)

else:

new\_snapshot = self.volume\_api.create\_snapshot(

context,

volume,

snapshot.get('display\_name'),

snapshot.get('description'),

\*\*kwargs)

......

在cinder.volume.api.API中，create\_snapshot\_force(...)和create\_snapshot(...)处理逻辑相同都是调用\_create\_snapshot(...)，只是入参force的传值不同。（这样将一个接口写成两个的用意可能是为了方便拓展？！）

def \_create\_snapshot(self, context,

volume, name, description,

force=False, metadata=None,

cgsnapshot\_id=None,

group\_snapshot\_id=None):

volume.assert\_not\_frozen()

# create\_snapshot\_in\_db(...)在数据库中创建一个快照记录，并且提交快照配额。

# 创建快照时需要检查volume的状态，以下状态的卷不能创建快照：

# 没有host信息的卷不能创建快照；

# maintenance状态的卷不能创建快照

# 正在迁移的卷不能创建快照

# replica卷不能创建快照

# 非force情形，只能对available卷创建快照

# 快照配额的预留和提交也由create\_snapshot\_in\_db(...)方法完成，配额涉及两个方面：

# 1.快照数目的配额；

# 2.快照磁盘占用的配额,通过CONF.no\_snapshot\_gb\_quota可以设定快照是否占用磁盘大小配额（默认是占用）

snapshot = self.create\_snapshot\_in\_db(

context, volume, name,

description, force, metadata, cgsnapshot\_id,

True, group\_snapshot\_id)

# 通知cinder-volume创建快照

self.volume\_rpcapi.create\_snapshot(context, volume, snapshot)

return snapshot

1. Cinder-Volume

Cinder-Volume相应创建快照的接口为create\_snapshot(...)该接口调用驱动的create\_snapshot(...)开始创建快照！在VolumeManager的create\_snapshot(...)方法中，没有对卷的状态加以检查，也就是说无论状态是否是in-use创建快照时Cinder不保证数据完整。创建热快照时数据是否完整可用，取决于Storage的驱动！

@objects.Snapshot.set\_workers

def create\_snapshot(self, context, snapshot):

"""Creates and exports the snapshot."""

context = context.elevated()

self.\_notify\_about\_snapshot\_usage(context, snapshot, "create.start")

try:

snapshot.context = context

# 调用驱动的create\_snapshot接口创建快照

model\_update = self.driver.create\_snapshot(snapshot)

# 更新快照信息

if model\_update:

snapshot.update(model\_update)

snapshot.save()

except Exception:

with excutils.save\_and\_reraise\_exception():

snapshot.status = fields.SnapshotStatus.ERROR

snapshot.save()

# 拷贝快照的metadata信息

......

self.\_notify\_about\_snapshot\_usage(context, snapshot, "create.end")

LOG.info(\_LI("Create snapshot completed successfully"),

resource=snapshot)

return snapshot.id

## Storage驱动快照接口

在Cinder中不同的设备驱动需要实现create\_snapshot(...)接口创建快照，且同时创建快照时需要保证数据结构的完成新。

cinder.volume.driver.VolumeDriver中的快照接口：

def create\_snapshot(self, snapshot):

"""Creates a snapshot."""

raise NotImplementedError()

1. NetApp ISCSI驱动快照实现

在NetApp中创建快照和从卷创建Clone使用的是同一个接口，也就是说在netapp看来，创建一个快照相当于从源卷创建一个Clone。

NetApp指定在Storage中，快照的命名格式为snapshot-XXXXX，卷的命名格式为volume-xxxxx。创建时使用的命令是volume file clone create，只指定了五个参数（vserver，volume，no-reserve，source-path，is-backup），其他参数均使用默认值。指定参数中-no-reserve的取值根据是否配置了瘦分配，is-backup参数的取值根据是否创建快照，创建快照时is-backup=true

# 创建卷Clone对应的Netapp Command

volume file clone create

# -vserver <vserver name> ：SVM路径

# -volume <volume name> ：源卷的Volume位置

# -source-path <path> : 源卷的Path

# -destination-path ：克隆卷的Path

# -is-backup {true|false} ：是否创建的是一个备份，创建备份时Clone卷不会拷贝源卷的divergence信息，默认时为True

# -snapshot-name|-s] <snapshot name> ：当源卷是个快照时，可以从快照创建克隆，注意netapp的快照是Volume级别的快照，而不是LUN级别的

# -range ：指定拷贝源卷的某些block到目的卷（不指定该参数时，拷贝整个LUN）

# -no-reserve ：是否不为Clone卷预留空间，默认情形下为True

# -ignore-streams ：在创建Clone的过程中是否忽略写入流（即创建过程中的写入是否合并到Clone卷中），默认为False

# -ignore-locks ：是否忽略源的lock状态

# -overwrite-destination ：直接将destination-path覆盖

# -qos-policy-group ：指定Clone卷的Qos

# -autodelete ：空间不足时是否删除Clone，默认为False

# -bypass-throttle ：是否忽略配额检查，默认为False

# -caching-policy ：指定Clone的缓存策略，指定该参数的同时需要指定-qos-policy-group，默认情形下为auto（该模式官方的说明是：Read caches all metadata and randomly read user data blocks, and write caches all randomly overwritten user data blocks.）

1. Huawei ISCSI驱动快照实现

当前华为存储5300，2650的快照功能均不可用，需要购买相应的功能授权。（注意：华为创建卷Clone实际上是基于卷Snapshot，因此该功能也不能使用！）

在5300设备上创建快照时，Request/Response信息如下：

Request URL: https://<ip>:8088/deviceManager/rest/<device\_id>/snapshot

Call Method: Nonex

# Request中PARENTID表示待创建卷的LUN\_ID,PARENTTYPE表示源的类型（11为LUN，27为快照）

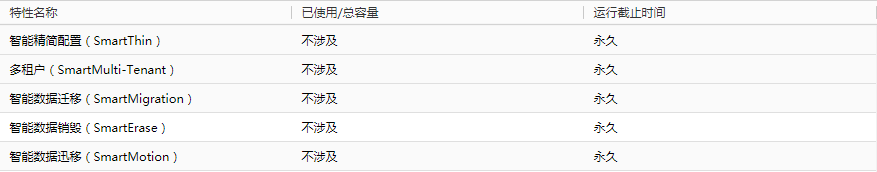
Request Data: {"PARENTTYPE": "11", "PARENTID": "38", "TYPE": "27", "NAME": "<snapshot\_id>", "DESCRIPTION": "<message>"}

Response Data:{"data":{},"error":{"code":1077937864,"description":"The snapshot license file is unavailable.","suggestion":"Import a valid license file."}}

根据Huawei产品文档，快照是一项增值特性，需要购买License才能使用。快照特性又叫***HyperSnap特性***。

参考产品文档华为的快照基本可以认为是源LUN的Clone，创建的快照LUN在激活之后是可以读写的，且产品文档中提到，当使用快照LUN还原源LUN时，一旦快照被写入过数据，那么就只能还原到当前的快照状态，而不能还原到快照创建时刻的状态。

当前5300设备拥有的License：



当前2650设备没有添加License！

1. NFS驱动的快照实现

***NFS快照原理***

在NFS驱动中磁盘以单个文件（假设为file0）的形式提供给虚拟机，在用户调用快照命令时，NFS驱动首先创建一个新的文件（假设为file1），接下来用户停止对file0的读写，将文件file1作为卷的读写文件。此时file0文件包含了创建快照时磁盘的状态，假设用户需要从快照还原磁盘，那么只要将磁盘的读写文件重新从file1指向到file0。

在Cinder通用的NFS驱动基本遵循了上述的快照原理，首次对NFS磁盘进行快照时，c-vol会在nfs\_shares\_config目录下创建一个快照映射文件volume-<id>.info，该文件存储了快照文件的映射顺序，格式如下：

{

"active": "volume-b3950e10-929a-4640-81a8-78d7cfd2f3f6.73cdd2fb-80cf-4870-a46d-c3b32564ba34",

"73cdd2fb-80cf-4870-a46d-c3b32564ba34": "volume-b3950e10-929a-4640-81a8-78d7cfd2f3f6.73cdd2fb-80cf-4870-a46d-c3b32564ba34",

"1f8766df-639f-4486-a7cf-602043bfde4f": "volume-b3950e10-929a-4640-81a8-78d7cfd2f3f6.1f8766df-639f-4486-a7cf-602043bfde4f",

"cee7ebe3-e994-4c64-a66a-2dcdeca57bd6": "volume-b3950e10-929a-4640-81a8-78d7cfd2f3f6.cee7ebe3-e994-4c64-a66a-2dcdeca57bd6"

}

上面的例子中，对卷b3950e10-929a-4640-81a8-78d7cfd2f3f6做了三次快照，每次快照都生成一个格式为volume-id.snapshot\_id的文件，该文件是卷的最新活动（actice）文件。即volume-id.snapshot\_id文件表示快照snapshot\_id创建完成后，对于volume-id的所有操作记录，volume-<id>.info的通用格式如下：

{

"active": "volume-id.snapshot\_id\_3",(\* changed!)

"snapshot\_id\_3": "volume-id.snapshot\_id\_3",

"snapshot\_id\_2": "volume-id.snapshot\_id\_2",(\* added!)

"snapshot\_id\_1": "volume-id.snapshot\_id\_1"，(\* added!)

}

***NFS创建available状态的卷快照***

接口函数：

@locked\_volume\_id\_operation

def create\_snapshot(self, snapshot):

"""Apply locking to the create snapshot operation."""

self.\_check\_snapshot\_support()

return self.\_create\_snapshot(snapshot)

在实际创建快照之前，Cinder需要判断卷的使用状态，in-use状态的磁盘，由于Nova正在读写数据，因此需要通知Nova进行额外的操作，保证数据一致性。

def \_create\_snapshot(self, snapshot):

"""Create a snapshot."""

......

# in-use状态的磁盘，由于Nova正在读写数据，因此需要通知Nova进行额外的操作，保证数据一致性

# \_create\_snapshot\_online和\_do\_create\_snapshot的入参含义如下：

# snapshot：将要创建的快照的数据库记录

# backing\_filename：当前磁盘读写文件（通过上文中提到的volume-<uuid>.info确定）

# new\_snap\_path：快照文件的文件名

if status == 'in-use':

self.\_create\_snapshot\_online(snapshot,

backing\_filename,

new\_snap\_path)

else:

self.\_do\_create\_snapshot(snapshot,

backing\_filename,

new\_snap\_path)

snap\_info['active'] = os.path.basename(new\_snap\_path)

snap\_info[snapshot.id] = os.path.basename(new\_snap\_path)

self.\_write\_info\_file(info\_path, snap\_info)

......

对于非in-use状态的卷，NFS驱动直接调用\_do\_create\_snapshot(...)创建一个快照，总结起来，该接口完成以下两个工作：

#1. 使用qemu-img命令创建快照文件（文件格式为qcow2）

# qemu-img create -f qcow2 -o backing\_file=<base\_file\_name>,backing\_fmt=<base\_file\_formate> <snapshot\_name> <snapshot\_size in GB>

#2. 指定<snapshot\_name>文件的后端为当前active的镜像（这里使用非安全模式进行合并）,

# qemu-img rebase -u -b <base\_file\_name> -F <base\_file\_formate> <snapshot\_name>

def \_do\_create\_snapshot(self, snapshot, backing\_filename,

new\_snap\_path):

......

# 获取当前磁盘NFS文件的信息，（主要是获取文件格式）

info = self.\_qemu\_img\_info(backing\_path\_full\_path,

snapshot.volume.name)

backing\_fmt = info.file\_format

command = ['qemu-img', 'create', '-f', 'qcow2', '-o',

'backing\_file=%s,backing\_fmt=%s' %

(backing\_path\_full\_path, backing\_fmt),

new\_snap\_path,

"%dG" % snapshot.volume.size]

self.\_execute(\*command, run\_as\_root=self.\_execute\_as\_root)

command = ['qemu-img', 'rebase', '-u',

'-b', backing\_filename,

'-F', backing\_fmt,

new\_snap\_path]

# qemu-img rebase must run as root for the same reasons as above

self.\_execute(\*command, run\_as\_root=self.\_execute\_as\_root)

NFS驱动调用\_do\_create\_snapshot(...)返回之后会修改快照映射文件volume-.info，之后整个创建快照的过程就完成了！

***NFS创建in-use状态的卷快照（热快照）***

创建热快照时，cinder需要通知Nova修改虚机的XML文件，并且通过libvirt切换磁盘的读写文件，以此保证数据的完整性。创建热快照时NFS驱动调用的接口是\_create\_snapshot\_online(...)：

# \_create\_snapshot\_online(...)会调用\_do\_create\_snapshot(...)创建一个空白的快照文件，非in-use状态的卷快照到这步就结束了。

# 对于热快照，需要Nova切换当前的磁盘读写文件，因此Cinder会调用Nova的API，并且等待Nova的操作完成（注意：这里Cinder等待的超时时间写死成600s）。

def \_create\_snapshot\_online(self, snapshot, backing\_filename,

new\_snap\_path):

# Perform online snapshot via Nova

context = snapshot.\_context

self.\_do\_create\_snapshot(snapshot,backing\_filename,new\_snap\_path)

connection\_info = {

'type': 'qcow2',

'new\_file': os.path.basename(new\_snap\_path),

'snapshot\_id': snapshot.id

}

try:

result = self.\_nova.create\_volume\_snapshot(context,snapshot.volume\_id,connection\_info)

LOG.debug('nova call result: %s', result)

except Exception:

LOG.exception(\_LE('Call to Nova to create snapshot failed'))

raise

# Loop and wait for result

# Nova will call Cinderclient to update the status in the database

# An update of progress = '90%' means that Nova is done

......

# Cinder 向nova发出的API请求如下：

http://<ip>:8774/v2.1/os-assisted-volume-snapshots

{

"snapshot": {

"create\_info": {

# 快照ID

"snapshot\_id": "d3a86deb-a104-4527-ad42-55e7c160deb9",

# 磁盘读写文件的格式

"type": "qcow2",

# 磁盘新的读写文件的文件名

"new\_file": "volume-08dde67a-8fe0-4a5e-9c81-ac50e2d61348.d3a86deb-a104-4527-ad42-55e7c160deb9"

},

# 磁盘ID

"volume\_id": "08dde67a-8fe0-4a5e-9c81-ac50e2d61348"

}

}

在nova的libvirt库中会根据Cinder提供的快照信息，更新VM的XML文件，并且调用libvirt的C语言函数库的snapshotCreateXML接口切换磁盘的读写文件。

def snapshot(self, conf, no\_metadata=False,

disk\_only=False, reuse\_ext=False, quiesce=False):

"""Creates a guest snapshot."""

......

device\_xml = conf.to\_xml()

if six.PY3 and isinstance(device\_xml, six.binary\_type):

device\_xml = device\_xml.decode('utf-8')

# 调用libvirt的C语言函数库这里self.\_domain是一个eventlet.tpool.Proxy类型，负责调用libvirt的C接口。

self.\_domain.snapshotCreateXML(device\_xml, flags=flags)

# 快照删除

## 接口分析

# 从删除快照时需要对快照加锁，防止删除正在使用中的快照

@coordination.synchronized('{snapshot.id}-{f\_name}')

def delete\_snapshot(self, context, snapshot, unmanage\_only=False):

......

try:

.......

# snapshot-delete命令和snapshot-unmanage命令共用delete\_snapshot(.....)

# snapshot-unmanage命令之将快照从OpenStack中删除，没有在后端实际删除数据

if unmanage\_only:

self.driver.unmanage\_snapshot(snapshot)

else:

self.driver.delete\_snapshot(snapshot)

except exception.SnapshotIsBusy:

......

# 删除之后需要对快照的配额进行回滚

try:

......

reservations = QUOTAS.reserve(context,project\_id=project\_id,\*\*reserve\_opts)

# 删除快照的数据库记录

snapshot.destroy()

# 提交配额更改

if reservations:

QUOTAS.commit(context, reservations, project\_id=project\_id)

## Storage驱动如何实现

对应的驱动接口函数是delete\_snapshot(...)

def delete\_snapshot(self, snapshot):

"""Driver entry point for deleting a snapshot."""

raise NotImplementedError()

1. NetApp ISCSI驱动

由于NetApp快照实际上就是一个LUN，因此删除快照和删除LUN操作操作相同。NetApp中删除卷的过程如下：

1.查询netapp驱动在内部维护一个字典（NetAppBlockStorageLibrary.lun\_table）,字典中通过 volume\_id/snapshot\_id 对 LUN的信息进行映射；

2.根据字典获取的metadata["Path"]，发送删除报文。

def delete\_snapshot(self, snapshot):

"""Driver entry point for deleting a snapshot."""

# 删除对应的LUN，这里\_delete\_lun(...)接口和删除卷的接口是同一个接口

self.\_delete\_lun(snapshot['name'])

LOG.debug("Snapshot %s deletion successful", snapshot['name'])

1. Huawei ISCSI驱动

华为存储删除过程和Netapp类似

def delete\_snapshot(self, snapshot):

snapshotname = huawei\_utils.encode\_name(snapshot.id)

volume\_name = huawei\_utils.encode\_name(snapshot.volume\_id)

.....

snapshot\_id = snapshot.provider\_location

if snapshot\_id is None:

snapshot\_id = self.client.get\_snapshot\_id\_by\_name(snapshotname)

if snapshot\_id and self.client.check\_snapshot\_exist(snapshot\_id):

# 将snapshot LUN脱机

self.client.stop\_snapshot(snapshot\_id)

# 删除对应的snapshot LUN

self.client.delete\_snapshot(snapshot\_id)

else:

LOG.warning(\_LW("Can't find snapshot on the array."))

1. NFS驱动

删除NFS快照时，需要将当前快照文件中的数据合并到他的Backend中，Cinder通过qemu-img commit命令实现这个功能。

qemu-img commit <snapshot\_del\_path>

删除NFS的快照时，可以分成两种情况：

情况一、删除顶层快照,即删除最后创建的快照

step1：将snapshot\_del的所有改动提交到他的backend

step2：将volume-.info文件中active image设为他的backend

step3：删除snapshot\_del文件

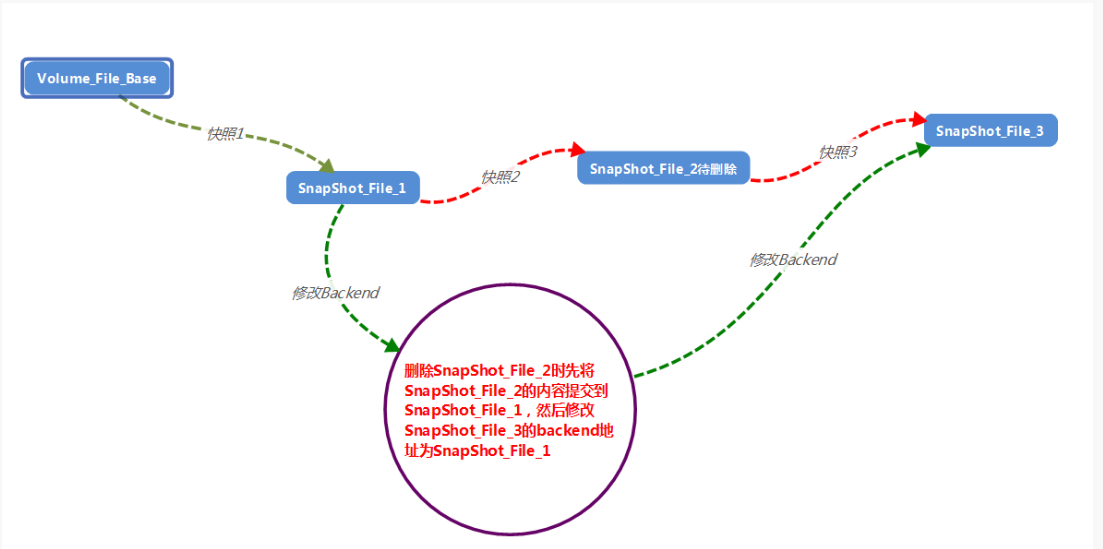
情况二、删除中间层快照

step1：将snapshot\_del的所有改动提交到他的backend

step2：将snapshot\_del顶层快照文件的backend，切换成他的backend

step3：删除snapshot\_del文件

下图展示了删除卷的第二层快照时的场景



# 从快照创建卷

从快照创建卷时，cinder有以下限制：

cinder create --snapshot-id <snapshot-id> <size>

#1.如果指定卷类型(通过--volume-type选项)，那么Cinder-API进行参数检查时要求指定的type类型和快照的type-类型（即快照源卷的type类型）中volume\_backend\_name的值一致；如果不一致时，要求当前环境只有一个service，即只有一个后端。

#2.如果不指定卷类型那么使用源卷的卷类型，如果源卷没有卷类型，那么使用默认卷类型，以上都不满足那么置空。

#3.默认情况下，从快照创建的卷和源卷位于同一个后端（即HOST取值完全相同），Cinder在创建卷时不通过cinder-scheduler，而是直接拷贝源卷的HOST。通过在Cinder\_API配置CONF.snapshot\_same\_host=False可以强行让源卷和创建的卷位于不同的HOST，但是这样storage的驱动会报错，创建卷的过程会失败！

#4.创建过程中Cinder\_volume会对快照资源加锁，防止在创建卷的过程中，快照被人删除。锁的名称为“<snapshot-id>delete\_snapshot”，锁的提供者是Tooz。

#5.如果指定的卷大小比源卷大，那么驱动先创建一个和源卷一样大小的LUN，之后掉用扩展方法，扩展到指定大小。

## 卷创建流程

在Cinder\_Volume的TaskFlow中CreateVolumeFromSpecTask完成通知Storage创建Volume的操作，该Task会根据不同的创建类型调用不同接口。snap类型时调用\_create\_from\_source\_volume(...)接口创建卷,该接口调用驱动的create\_volume\_from\_snapshot接口和Storage交互

对应的日志：



# CreateVolumeFromSpecTask的execute方法：

def execute(self, context, volume, volume\_spec):

......

# 根据type值调用接口创建Volume

create\_type = volume\_spec.pop('type', None)

......

elif create\_type == 'snap':

model\_update = self.\_create\_from\_snapshot(context, volume,\*\*volume\_spec)

......

# \_create\_from\_snapshot方法:

def \_create\_from\_snapshot(self, context, volume, snapshot\_id,

\*\*kwargs):

# 调用驱动的create\_volume\_from\_snapshot(...)接口

snapshot = objects.Snapshot.get\_by\_id(context, snapshot\_id)

model\_update = self.driver.create\_volume\_from\_snapshot(volume, snapshot)

## 不同驱动如何实现

1. NetApp的ISCSI驱动

由于在netapp中快照实际上就是一个卷Clone，所以对NetApp来说从快照创建卷实际上就是从快照对应的LUN再拷贝一份。

# 在NetApp驱动中Clone和从卷创建调用的是同一个接口：

# 从卷Snapshot创建卷

def create\_volume\_from\_snapshot(self, volume, snapshot):

source = {'name': snapshot['name'], 'size': snapshot['volume\_size']}

return self.\_clone\_source\_to\_destination(source, volume)

# 从卷创建卷

def create\_cloned\_volume(self, volume, src\_vref):

src\_lun = self.\_get\_lun\_from\_table(src\_vref['name'])

source = {'name': src\_lun.name, 'size': src\_vref['size']}

return self.\_clone\_source\_to\_destination(source, volume)

# \_clone\_source\_to\_destination(......)接口调用的是volume file clone create命令（和创建快照是同一个命令，前面分析过！）。

1. Huawei的ISCSI驱动

由于无法创建华为存储的快照，只能通过代码分析。从驱动的create\_volume\_from\_snapshot(...)接口来看，Huawei存储从快照创建卷基于的是Copy而不是Clone，且驱动中说明创建卷的时间正比于卷的大小！

1. NFS驱动

假设某个卷创建了三个快照，如下图，现在需要从Snapshot2创建一个新卷，根据NFS的快照原理，New\_Volume\_File需要将Volume\_File\_Base文件和SnapShot\_File\_1文件的内容拷贝出来，NFS进行拷贝时使用的命令是qemu-img convert，执行完成后Volume\_File\_Base和SnapShot\_File\_1的内容被整合到New\_Volume\_File，该文件此后能够作为一个独立的NFS磁盘文件使用。

由于NFS驱动从快照创建磁盘时是基于qemu-img convert，该命令是进行拷贝的，因此速度和文件大小成正比（在我的环境上1G大小的卷，需要12s）：

qemu-img convert -O raw <snap\_volume\_file\_path> <new\_volume\_file\_path>

