# Ironic 状态机

参考：[Bare Metal State Machine — ironic 21.5.0.dev115 documentation (openstack.org)](https://docs.openstack.org/ironic/latest/user/states.html#states)

这个状态在源码中叫做 provision state。Ironic 中的状态分为两种类型：稳定状态和临时状态。稳定状态是在没有外部主动触发动作的情况下（例如调用 api 修改期望状态）保持不变，而临时状态是在从一个稳定状态向另外一个稳定状态转换过程中所经过的状态，ironic 代码会不断修改临时状态，成功时到达下一个稳定状态，失败时回滚到初始稳定状态。

重点是几个稳定状态：

* enroll：node 创建时的初始状态，表示这个 node 注册到了 ironic。Enroll 状态可以通过 manage 动作触发进入 verifying
* manageable：有很多种情况处于 manageable，不同情况下node 的实际状态（信息，数据）不一样
* available：表示 node 已经被配置和清理，准备好部署了
* active：部署成功，交付用户运行
* error：部署回滚失败
* rescue：有一个 rescue ramdisk 在运行。

## 如何驱动 ironic 的状态流转？

通过以下 api：

|  |
| --- |
| /v1/nodes/{node\_ident}/states/provision |

参见：[Bare Metal API — Ironic API Reference documentation (openstack.org)](https://docs.openstack.org/api-ref/baremetal/?expanded=change-node-provision-state-detail)

设置其中 target 字段为期望的状态，会触发状态机流转。

下面看下这个 api 的处理流程：

NodeStatesController.provision()

|  |
| --- |
| (ironic/api/controllers/v1/node.py)  def provision(self, node\_ident, target, configdrive=None,  clean\_steps=None, deploy\_steps=None,  rescue\_password=None, disable\_ramdisk=None):  …  m = ir\_states.machine.copy()  m.initialize(rpc\_node.provision\_state)  …  # 调用 \_do\_provision\_action()  self.\_do\_provision\_action(rpc\_node, target, configdrive, clean\_steps,  deploy\_steps, rescue\_password,  disable\_ramdisk) |

NodeStatesController. \_do\_provision\_action()

|  |
| --- |
| (ironic/api/controllers/v1/node.py)  def \_do\_provision\_action(self, rpc\_node, target, configdrive=None,  clean\_steps=None, deploy\_steps=None,  rescue\_password=None, disable\_ramdisk=None):  …  # PROVISION\_ACTION\_STATES 为 [‘manage’, ‘provide’, ‘abort’, ‘adopt’]  elif target in PROVISION\_ACTION\_STATES:  api.request.rpcapi.do\_provisioning\_action(  api.request.context, rpc\_node.uuid, target, topic) |

ConductorManager.do\_provisioning\_action()

|  |
| --- |
| (ironic/conductor/manager.py)  # 其中，action 为 api 中指定的 target  def do\_provisioning\_action(self, context, node\_id, action):  …  if (action == states.VERBS['manage']  and node.provision\_state == states.ENROLL):  task.process\_event(  'manage',  callback=self.\_spawn\_worker,  call\_args=(verify.do\_node\_verify, task),  err\_handler=utils.provisioning\_error\_handler)  return |

do\_provisioning\_action() 中初始化一个异步的 worker 进行处理，调用do\_node\_verify()。

# Ironic 中的 Interface

* BIOSInterface：操作裸金属 BIOS 配置
* BootInterface：操作裸金属启动相关的配置
* ConsoleInterface：实现裸金属服务的 Console 连接
* DeployInterface：负责用户镜像的部署
* InspectInterface：负责 inspect，即硬件信息的收集
* ManagementInterface：负责裸金属远程管理，如 impi 接口的控制
* NetworkInterface：负责裸金属的网络配置
* PowerInterface：负责裸金属的电源管理，如获取电源状态，关机等
* RAIDInterface：负责裸金属磁盘 Raid 的配置
* RescueInterface：负责 Rescue 操作
* StorageInterface：负责存储操作 boot from volume 机制
* VendorInterface：负责厂商信息相关的操作

Ironic 对每一项接口拆分的比较细，实际上 PowerInterface 也可以算作 ManagementInterface 的一部分。

## BootInterface

目前典型的实现：

* PXEBoot：以 pxe 方式启动
* iPXEBoot：以 ipxe 方式启动
* DracRedfishVirtualMediaBoot：Drac 虚拟介质启动

## DeployInterface

目前典型的实现：

* AgentDeploy：即和 ironic-python-agent 配合工作
* AnsibleDeploy：基于 ansible

部署流程。

### 1 api 修改 ProvisionState 为 ACTIVE，ironic-api 处理

|  |
| --- |
| (ironic/api/controllers/v1/node.py)  def \_do\_provision\_action(self, rpc\_node, target, configdrive=None,  clean\_steps=None, deploy\_steps=None,  rescue\_password=None, disable\_ramdisk=None):  …  if target in (ir\_states.ACTIVE, ir\_states.REBUILD, ir\_states.DEPLOY):  rebuild = (target == ir\_states.REBUILD)  if deploy\_steps:  \_check\_deploy\_steps(deploy\_steps)  api.request.rpcapi.do\_node\_deploy(context=api.request.context,  node\_id=rpc\_node.uuid,  rebuild=rebuild,  configdrive=configdrive,  topic=topic,  deploy\_steps=deploy\_steps) |

### 2 ironic-api 调用 conductor 的 do\_node\_deploy() 的 rpc，rpc 中处理

|  |
| --- |
| （ironic/conductor/manager.py）  def do\_node\_deploy(self, context, node\_id, rebuild=False,  configdrive=None, deploy\_steps=None):  …  with task\_manager.acquire(context, node\_id, shared=False,  purpose='node deployment') as task:  deployments.validate\_node(task, event=event)  deployments.start\_deploy(task, self, configdrive, event=event,  deploy\_steps=deploy\_steps) |

### 3 conductor 中启动一个异步worker，处理部署过程

|  |
| --- |
| （ironic/conductor/deployments.py）  def start\_deploy(task, manager, configdrive=None, event='deploy',  deploy\_steps=None):  …  try:  task.process\_event(  event,  callback=manager.\_spawn\_worker,  call\_args=(do\_node\_deploy, task,  manager.conductor.id, configdrive, deploy\_steps),  err\_handler=utils.provisioning\_error\_handler)  … |

### 4异步worker 中调用 DeployInterface.prepare()，然后执行后续部署步骤

|  |
| --- |
| （ironic/conductor/deployments.py）  def do\_node\_deploy(task, conductor\_id=None, configdrive=None,  deploy\_steps=None):  …  try:  task.driver.deploy.prepare(task)  …  do\_next\_deploy\_step(task, 0) |

### 5 do\_next\_deploy\_step() 中依次执行定义的deploy\_step

|  |
| --- |
| （ironic/conductor/deployments.py）  def do\_next\_deploy\_step(task, step\_index):  """Do deployment, starting from the specified deploy step.  :param task: a TaskManager instance with an exclusive lock  :param step\_index: The first deploy step in the list to execute. This  is the index (from 0) into the list of deploy steps in the node's  driver\_internal\_info['deploy\_steps']. Is None if there are no steps  to execute.  """  node = task.node  def \_iter\_steps():  if step\_index is None:  return # short-circuit to the end  idx = step\_index  # The list can change in-flight, do not cache it!  while idx < len(node.driver\_internal\_info['deploy\_steps']):  yield idx, node.driver\_internal\_info['deploy\_steps'][idx]  idx += 1  # Execute each step until we hit an async step or run out of steps, keeping  # in mind that the steps list can be modified in-flight.  for idx, step in \_iter\_steps():  node.deploy\_step = step  # deploy\_step\_index 表示当前执行的步骤  node.set\_driver\_internal\_info('deploy\_step\_index', idx)  node.save()  interface = getattr(task.driver, step.get('interface'))  LOG.info('Executing %(step)s on node %(node)s',  {'step': step, 'node': node.uuid})  try:  result = interface.execute\_deploy\_step(task, step) |

#### deploy\_steps

从上面的流程可以看到，部署的核心在于 deploy steps，这里看下 deploy step 是如何定义的。

从 do\_next\_deploy\_step() 中看到 deploy\_steps 从 node.driver\_internal\_info['deploy\_steps'] 中取出，其设置是在 do\_node\_deploy() 中（prepare() 之后）：

|  |
| --- |
| （ironic/conductor/deployments.py）  def do\_node\_deploy(task, conductor\_id=None, configdrive=None,  deploy\_steps=None):  …  conductor\_steps.set\_node\_deployment\_steps(task, skip\_missing=True) |

|  |
| --- |
| (ironic/conductor/steps.py)  def set\_node\_deployment\_steps(task, reset\_current=True, skip\_missing=False):  """Set up the node with deployment step information for deploying.  Get the deploy steps from the driver.  :param reset\_current: Whether to reset the current step to the first one.  :raises: InstanceDeployFailure if there was a problem getting the  deployment steps.  """  node = task.node  node.set\_driver\_internal\_info('deploy\_steps', \_get\_all\_deployment\_steps(  task, skip\_missing=skip\_missing))  LOG.debug('List of the deploy steps for node %(node)s: %(steps)s', {  'node': node.uuid,  'steps': node.driver\_internal\_info['deploy\_steps']  })  if reset\_current:  node.deploy\_step = {}  node.set\_driver\_internal\_info('deploy\_step\_index', None)  node.save() |

https://docs.openstack.org/ironic/latest/admin/node-deployment.html

关于 deploy steps 的几点：

（1）priority 为 0 的 deploy step 不生效

（2）deploy template 通过 trait 与 node 和 flavor 关联生效

（3）deploy steps 的设置有几个途径：（1）在触发 deploy 操作的时候提供，如：

baremetal node deploy <node> \

--deploy-steps '[{"interface": "bios", "step": "apply\_configuration", "args": {"settings": [{"name": "LogicalProc", "value": "Enabled"}]}, "priority": 150}]'

这种称为 user\_steps，保存在 node.driver\_internal\_info 的 user\_deploy\_steps 字段中。这是在 do\_node\_deploy 函数中处理的：

（ironic/conductor/deployments.py）

try:

# If any deploy steps provided by user, save them to node. They will be

# validated & processed later together with driver and deploy template

# steps.

if deploy\_steps:

node.set\_driver\_internal\_info('user\_deploy\_steps', deploy\_steps)

node.save()

（2）通过 deploy template 模板提供，参考 https://docs.openstack.org/ironic/latest/admin/node-deployment.html。

（3）通过代码中 driver 的 interface 提供的，例如 @base.deploy\_step 加上这个装饰器，对应的函数就会变成一个 deploy step。

其中，用户提供的 deploy step 会覆盖 driver 和模板中的。

下面通过 \_get\_all\_deployment\_steps 这个函数看下从这几个地方获取 deploy step 的过程。

## \_get\_all\_deployment\_steps

（ironic/conductor/steps.py）

def \_get\_all\_deployment\_steps(task, skip\_missing=False):

"""Get deployment steps for task.node.

Deployment steps from matching deployment templates are combined with those

from driver interfaces and all enabled steps returned in priority order.

:param task: A TaskManager object

:raises: InstanceDeployFailure if there was a problem getting the

deploy steps.

:returns: A list of deploy step dictionaries

"""

# Get deploy steps provided by user via argument if any. These steps

# override template and driver steps when overlap.

# 获取并验证用户提供的 deploy steps

user\_steps = \_get\_validated\_user\_deploy\_steps(

task, skip\_missing=skip\_missing)

# Gather deploy steps from deploy templates and validate.

# NOTE(mgoddard): although we've probably just validated the templates in

# do\_node\_deploy, they may have changed in the DB since we last checked, so

# validate again.

# 获取并验证模板中的 deploy steps

template\_steps = \_get\_validated\_steps\_from\_templates(

task, skip\_missing=skip\_missing)

# Take only template steps that are not already provided by user

# 覆盖模板中的

user\_step\_keys = {(s['interface'], s['step']) for s in user\_steps}

new\_template\_steps = [s for s in template\_steps

if (s['interface'], s['step']) not in user\_step\_keys]

user\_steps.extend(new\_template\_steps)

# Gather enabled deploy steps from drivers.

# 获取 driver 中的

driver\_steps = \_get\_deployment\_steps(task, enabled=True, sort=False)

# Remove driver steps that have been disabled or overridden by user steps.

# 覆盖 driver 中的

user\_step\_keys = {(s['interface'], s['step']) for s in user\_steps}

steps = [s for s in driver\_steps

if (s['interface'], s['step']) not in user\_step\_keys]

# Add enabled user steps.

# priority 为0 的 step 不生效

enabled\_user\_steps = [s for s in user\_steps if s['priority'] > 0]

steps.extend(enabled\_user\_steps)

# 对 deploy step 进行排序

return \_sorted\_steps(steps, \_deploy\_step\_key)

下面重点看下 driver 中预定义的 deploy step 是如何获取的，参考上文，入口函数是 \_get\_deployment\_steps：

（ironic/conductor/steps.py）

def \_get\_deployment\_steps(task, enabled=False, sort=True):

"""Get deployment steps for task.node.

:param task: A TaskManager object

:param enabled: If True, returns only enabled (priority > 0) steps. If

False, returns all deploy steps.

:param sort: If True, the steps are sorted from highest priority to lowest

priority. For steps having the same priority, they are sorted from

highest interface priority to lowest.

:raises: InstanceDeployFailure if there was a problem getting the

deploy steps.

:returns: A list of deploy step dictionaries

"""

sort\_key = \_deploy\_step\_key if sort else None

return \_get\_steps(task, DEPLOYING\_INTERFACE\_PRIORITY, 'get\_deploy\_steps',

enabled=enabled, sort\_step\_key=sort\_key)

调用 \_get\_steps() 函数：

（ironic/conductor/steps.py）

// interfaces 为 interface 的列表，前面传入的是 DEPLOYING\_INTERFACE\_PRIORITY

// get\_method 表示调用 interface 的哪个函数获取 deploy steps，前面传入的是 'get\_deploy\_steps'

def \_get\_steps(task, interfaces, get\_method, enabled=False,

sort\_step\_key=None, prio\_overrides=None):

"""Get steps for task.node.

:param task: A TaskManager object

:param interfaces: A dictionary of (key) interfaces and their

(value) priorities. These are the interfaces that will have steps of

interest. The priorities are used for deciding the priorities of steps

having the same priority.

:param get\_method: The method used to get the steps from the node's

interface; a string.

:param enabled: If True, returns only enabled (priority > 0) steps. If

False, returns all steps.

:param sort\_step\_key: If set, this is a method (key) used to sort the steps

from highest priority to lowest priority. For steps having the same

priority, they are sorted from highest interface priority to lowest.

:param prio\_overrides: An optional dictionary of priority overrides for

steps, e.g:

{'deploy.erase\_devices\_metadata': '123',

'management.reset\_bios\_to\_default': '234'}

:raises: NodeCleaningFailure or InstanceDeployFailure if there was a

problem getting the steps.

:returns: A list of step dictionaries

"""

# Get steps from each interface

steps = list()

for interface in interfaces:

interface = getattr(task.driver, interface)

if interface:

# NOTE(janders) get all steps to start with, regardless of whether

# enabled is True and priority is zero or not; we need to apply

# priority overrides prior to filtering out disabled steps

interface\_steps = [x for x in getattr(interface, get\_method)(task)]

steps.extend(interface\_steps)

# Iterate over steps to apply prio overrides if set

if prio\_overrides is not None:

for step in steps:

override\_key = '%(interface)s.%(step)s' % step

override\_value = prio\_overrides.get(override\_key)

if override\_value:

step["priority"] = int(override\_value)

# NOTE(janders) If enabled is set to True, we filter out steps with zero

# priority now, after applying priority overrides

if enabled:

steps = [x for x in steps if not (x.get('priority') == 0)]

if sort\_step\_key:

steps = \_sorted\_steps(steps, sort\_step\_key)

return steps

所以，核心在于 interface 的 get\_deploy\_steps 函数，这定义在 interface 的基类 BaseInterface 中：

（ironic/drivers/base.py）

def get\_deploy\_steps(self, task):

"""Get a list of (enabled and disabled) deploy steps for the interface.

This function will return all deploy steps (both enabled and disabled)

for the interface, in an unordered list.

:param task: A TaskManager object, useful for interfaces overriding

this function

:raises InstanceDeployFailure: if there is a problem getting the steps

from the driver. For example, when a node (using an agent driver)

has just been enrolled and the agent isn't alive yet to be queried

for the available deploy steps.

:returns: A list of deploy step dictionaries

"""

return self.deploy\_steps

可以看到，返回的是 self.deploy\_steps。其初始化在 BaseInterface 的 \_\_new\_\_ 方法中：

（ironic/drivers/base.py）

def \_\_new\_\_(cls, \*args, \*\*kwargs):

# Get the list of clean steps, deploy steps and verify steps when the

# interface is initialized by the conductor.

# We use \_\_new\_\_ instead of \_\_init\_\_\_

# to avoid breaking backwards compatibility with all the drivers.

# We want to return all steps, regardless of priority.

super\_new = super(BaseInterface, cls).\_\_new\_\_

if super\_new is object.\_\_new\_\_:

instance = super\_new(cls)

else:

instance = super\_new(cls, \*args, \*\*kwargs)

instance.clean\_steps = []

instance.deploy\_steps = []

instance.verify\_steps = []

for n, method in inspect.getmembers(instance, inspect.ismethod):

if getattr(method, '\_is\_clean\_step', False):

# Create a CleanStep to represent this method

step = {'step': method.\_\_name\_\_,

'priority': method.\_clean\_step\_priority,

'abortable': method.\_clean\_step\_abortable,

'argsinfo': method.\_clean\_step\_argsinfo,

'interface': instance.interface\_type,

'requires\_ramdisk':

method.\_clean\_step\_requires\_ramdisk}

instance.clean\_steps.append(step)

if getattr(method, '\_is\_deploy\_step', False):

# Create a DeployStep to represent this method

step = {'step': method.\_\_name\_\_,

'priority': method.\_deploy\_step\_priority,

'argsinfo': method.\_deploy\_step\_argsinfo,

'interface': instance.interface\_type}

instance.deploy\_steps.append(step)

if getattr(method, '\_is\_verify\_step', False):

# Create a VerifyStep to represent this method

step = {'step': method.\_\_name\_\_,

'priority': method.\_verify\_step\_priority,

'interface': instance.interface\_type}

instance.verify\_steps.append(step)

if instance.clean\_steps:

LOG.debug('Found clean steps %(steps)s for interface '

'%(interface)s',

{'steps': instance.clean\_steps,

'interface': instance.interface\_type})

if instance.deploy\_steps:

LOG.debug('Found deploy steps %(steps)s for interface '

'%(interface)s',

{'steps': instance.deploy\_steps,

'interface': instance.interface\_type})

if instance.verify\_steps:

LOG.debug('Found verify steps %(steps)s for interface '

'%(interface)s',

{'steps': instance.deploy\_steps,

'interface': instance.interface\_type})

可以看到，它会遍历 interface 的所有方法，如果某个方法带有 \_is\_deploy\_step 属性，就认为这个方法是一个 deploy step 实现，将其转换为一个 deploy step，其中，方法名为 step 的名称。

为此，代码中实现了一个 deploy\_step 的装饰器，通过这个装饰器装饰的函数，会加上 \_is\_deploy\_step 属性：

（ironic/drivers/base.py）

def deploy\_step(priority, argsinfo=None):

"""Decorator for deployment steps.

Only steps with priorities greater than 0 are used.

These steps are ordered by priority from highest value to lowest

value. For steps with the same priority, they are ordered by driver

interface priority (see conductor.steps.DEPLOYING\_INTERFACE\_PRIORITY).

execute\_deploy\_step() will be called on each step.

Decorated deploy steps must take as the only positional argument, a

TaskManager object.

Deploy steps can be either synchronous or asynchronous. If the step is

synchronous, it should return `None` when finished, and the conductor

will continue on to the next step. While the deploy step is executing, the

node will be in `states.DEPLOYING` provision state. If the step is

asynchronous, the step should return `states.DEPLOYWAIT` to the

conductor before it starts the asynchronous work. When the step is

complete, the step should make an RPC call to `continue\_node\_deploy` to

move to the next step in deployment. The node will be in

`states.DEPLOYWAIT` provision state during the asynchronous work.

Examples::

class MyInterface(base.BaseInterface):

@base.deploy\_step(priority=100)

def example\_deploying(self, task):

# do some deploying

:param priority: an integer (>=0) priority; used for determining the order

in which the step is run in the deployment process.

:param argsinfo: a dictionary of keyword arguments where key is the name of

the argument and value is a dictionary as follows::

'description': <description>. Required. This should include

possible values.

'required': Boolean. Optional; default is False. True if this

argument is required. If so, it must be specified in

the deployment request; false if it is optional.

:raises InvalidParameterValue: if any of the arguments are invalid

"""

def decorator(func):

func.\_is\_deploy\_step = True

if isinstance(priority, int) and priority >= 0:

func.\_deploy\_step\_priority = priority

else:

raise exception.InvalidParameterValue(

\_('"priority" must be an integer value >= 0, instead of "%s"')

% priority)

\_validate\_argsinfo(argsinfo)

func.\_deploy\_step\_argsinfo = argsinfo

return func

return decorator

这就是 https://docs.openstack.org/ironic/latest/contributor/deploy-steps.html 中用 @base.deploy\_step 装饰器实现自定义 deploy step 的原理。

## \_sorted\_steps

对 deploy steps 进行排序：

（ironic/conductor/steps.py）

// 参数中，steps 为一个 list，其中每个元素是一个 dict，格式参考 https://docs.openstack.org/ironic/latest/admin/node-deployment.html

// {

"interface": "<name of the driver interface>",

"step": "<name of the step>",

"args": {

"<arg1>": "<value1>",

"<arg2>": "<value2>"

},

"priority": <priority of the step>

//}

// sort\_step\_key 表示按哪个字段排序

def \_sorted\_steps(steps, sort\_step\_key):

"""Return a sorted list of steps.

:param sort\_step\_key: If set, this is a method (key) used to sort the steps

from highest priority to lowest priority. For steps having the same

priority, they are sorted from highest interface priority to lowest.

:returns: A list of sorted step dictionaries.

"""

# Sort the steps from higher priority to lower priority

return sorted(steps, key=sort\_step\_key, reverse=True)

关于 sort\_step\_key，前面 \_get\_all\_deployment\_steps() 函数中传入的是 \_deploy\_step\_key：

（ironic/conductor/steps.py）

def \_deploy\_step\_key(step):

"""Sort by priority, then interface priority in event of tie.

:param step: deploy step dict to get priority for.

"""

return (step.get('priority'),

DEPLOYING\_INTERFACE\_PRIORITY[step.get('interface')])

可以看到，先按 step 中指定的优先级进行排序，相同优先级下，按预定义的 interface 进行排序。

### 总结

部署过程中对 DeployInterface 的调用顺序：

1. validate()，在 start\_deloy() 中启动异步 worker 之前，先对部署信息进行验证
2. prepare()，在 do\_node\_deploy() 中执行 deploy steps 之前，做部署的准备
3. 执行各个 deploy steps，deploy steps 由 DeployInterface 通过 @base.deploy\_step 装饰器定义，按优先级 priority 排序，direct 插件的 deploy steps：

* CustomAgentDeploy.deploy() （priority=100）：prepare 阶段准备好了带有 ipa 的 ramdisk 启动，这里重启通过 pxe 进入 ipa 系统
* AgentDeploy.write\_image() (priority=80）：调用 ipa 下载用户镜像，写入磁盘
* CustomAgentDeploy.prepare\_instance\_boot() (priority=60）：清理 pxe 配置，设置从 disk 启动
* AgentDeployMixin.tear\_down\_agent()（priority=40)：power off，关闭 ipa 系统
* AgentOobStepsMixin.switch\_to\_tenant\_network()（priority=30)：退出部署网络，切换到租户网络
* AgentOobStepsMixin.boot\_instance() （priority=20)：启动进入用户系统

部署完成。

## NetworkInterface

NetworkInterface 负责裸金属网络的实现，目前有以下几个插件：

* FlatNetwork：基于 Neutron的扁平网络实现，不具有多租户能力
* NeutronNetwork：基于 Neutron 的overlay 网络实现，具有多租户能力

### Ironic 网络配置的流程

参考：[ironic与多租户网络支持（二） (sohu.com)](https://www.sohu.com/a/133932335_468741)

1. 租户网络的vif 创建，这是在 nova 的创建实例的流程中，通过 ironic virt driver 调用 ironic 的 vif\_attach()，其实际做的事情很简单，最重要的是把绑定到 vif 的 ironic port 加上 TENANT\_VIF\_KEY 的信息，后面会使用该信息。此时，并没有实际对交换机做配置，实际的配置在切换网络的过程中。
2. 切换网络，在切换网络时，如果切换到非租户网络，如部署网络，ironic 会自动调用 neutron 在对应网络下创建新的vif，并与 ironic port 进行绑定
3. 当部署完成，切换到租户网络时，plug\_port\_to\_tenant\_network() 中会根据 TENANT\_VIF\_KEY 找到绑定到vif 的 ironic port，更新其binding:host\_id 和 binding:profile，其中，binding:profile 中记录了交换机的信息。当其调用 neutron update port 接口时，neutron 会调用BaremetalMechanismDriver插件的 update\_port\_postcommit()，在其中会调用交换机的接口进行配置。

### vif\_attach

即把虚拟网卡的配置信息映射到裸金属的物理网卡上。具体有两个流程会调用。

#### 实例创建过程中

1. nova 调用 neutron 创建虚拟网卡

实例创建过程中，nova 会调用 neutron 的 api，创建网络相关内容，如 port，具体是在：

|  |
| --- |
| （nova/compute/manager.py）  @contextlib.contextmanager  def \_build\_resources(self, context, instance, requested\_networks,  security\_groups, image\_meta, block\_device\_mapping,  resource\_provider\_mapping, accel\_uuids):  …  try:  …  # 调用 neutron 创建 port  network\_info = self.\_build\_networks\_for\_instance(context, instance,  requested\_networks, security\_groups,  resource\_provider\_mapping, network\_arqs)  resources['network\_info'] = network\_info  …  try:  …  # Depending on a virt driver, some network configuration is  # necessary before preparing block devices.  # 调用 ironic driver 把虚拟网卡插到裸金属上  self.driver.prepare\_networks\_before\_block\_device\_mapping(  instance, network\_info)  …  … |

其中，prepare\_networks\_before\_block\_device\_mapping() 是我们这里要关注的，看下 ironic driver 的实现。

1. 调用 ironic vif\_attach

|  |
| --- |
| （nova/nova/virt/ironic/driver.py）  def prepare\_networks\_before\_block\_device\_mapping(self, instance,  network\_info):  …  try:  node = self.\_get\_node(instance.node)  self.\_plug\_vifs(node, instance, network\_info)  …  def \_plug\_vifs(self, node, instance, network\_info):  …  for vif in network\_info:  port\_id = str(vif['id'])  self.\_plug\_vif(node, port\_id)  def \_plug\_vif(self, node, port\_id):  last\_attempt = 5  for attempt in range(0, last\_attempt + 1):  try:  self.ironicclient.call("node.vif\_attach", node.uuid,  port\_id, retry\_on\_conflict=False) |

这里调用了 ironic 的 node.vif\_attach 接口。在 ironic 的 node.vif\_attach() 接口中，调用了 NetworkInterface 的 vif\_attach：

|  |
| --- |
| （ironic/conductor/manager.py）  def vif\_attach(self, context, node\_id, vif\_info):  …  with task\_manager.acquire(context, node\_id,  purpose='attach vif') as task:  task.driver.network.validate(task)  # 调用 network 插件的 vif\_attach  task.driver.network.vif\_attach(task, vif\_info) |

|  |
| --- |
| (ironic/ironic/drivers/modules/network/common.py)  def vif\_attach(self, task, vif\_info):  …  port\_like\_obj = get\_free\_port\_like\_object(  task, vif\_id, physnets, vif\_info)  …  # Address is optional for portgroups  if port\_like\_obj.address:  try:  neutron.update\_port\_address(vif\_id, port\_like\_obj.address,  context=task.context)  except exception.FailedToUpdateMacOnPort:  raise exception.NetworkError(\_(  "Unable to attach VIF %(vif)s because Ironic can not "  "update Neutron port %(port)s MAC address to match "  "physical MAC address %(mac)s") % {  'vif': vif\_id, 'port': vif\_id,  'mac': port\_like\_obj.address})  self.\_save\_vif\_to\_port\_like\_obj(port\_like\_obj, vif\_id) |

vif\_attach() 所做的操作，就是把 vif 和 node 的 port 或 portgroup 绑定，更新 neutron port 的 mac 地址。

### 网络切换

Todo

### 问题一，网络信息是如何传递给 ironic 的？

Ironic 中使用的网络信息包含几部分：

* 服务器本身的物理网卡信息，体现在 ironic port 中，在 inspect 阶段通过人为创建或 ironic inspector 创建，具体配置时，ironic 会把一个 ironic port 和一个vif 进行绑定
* 租户网络信息，即 vif 信息，由 nova 创建，ironic 会把 vif 和 ironic port 进行绑定。
* 租户网络、部署网络等使用的网络，通过配置文件指定 neutron network 的 id，ironic 会通过网络插件自动进行网络切换，并且，当使用非租户网络时，还会自动创建 neutron port（vif）并和 ironic port 进行绑定，使用完之后进行删除。

这里以部署阶段倒数第二步 switch\_to\_tenant\_network() 为例，看下 ironic 中网络切换的过程。

|  |
| --- |
| （ironic/drivers/modules/agent\_base.py）  @METRICS.timer('AgentOobStepsMixin.switch\_to\_tenant\_network')  @base.deploy\_step(priority=30)  @task\_manager.require\_exclusive\_lock  def switch\_to\_tenant\_network(self, task):  """Deploy step to switch the node to the tenant network.  :param task: a TaskManager object containing the node  """  try:  with manager\_utils.power\_state\_for\_network\_configuration(task):  task.driver.network.remove\_provisioning\_network(task)  task.driver.network.configure\_tenant\_networks(task)  except Exception as e:  … |

先调用 remove\_provisioning\_network() 从部署网络退出，然后调用 configure\_tenant\_networks() 切换到租户网络。

接下来看下 NeutronNetwork 插件的实现。

#### NeutronNetwork.remove\_provisioning\_network()

|  |
| --- |
| （ironic/drivers/modules/network/neutron.py）  def remove\_provisioning\_network(self, task):  """ """  # self.get\_provisioning\_network\_uuid() 获取配置文件中指定的 provisioning 网络所属的 neutron network id  return self.\_remove\_network(  task, self.get\_provisioning\_network\_uuid(task), 'provisioning') |

\_remove\_network() 是一个公共函数，退出网络都调用这个函数实现。

NeutronNetwork.\_remove\_network()

|  |
| --- |
| （ironic/drivers/modules/network/neutron.py）  # 参数中，network 为要退出的 neutron network id  # process 用于删除 ironic node 的 internal\_info 中记录的vif信息，例如这里是  # provisioning\_vif\_port\_id  def \_remove\_network(self, task, network, process):  LOG.info('Removing ports from %s network for node %s',  process, task.node.uuid)  neutron.remove\_ports\_from\_network(task, network)  field = '%s\_vif\_port\_id' % process  for port in task.ports:  if field in port.internal\_info:  internal\_info = port.internal\_info  del internal\_info[field]  port.internal\_info = internal\_info  port.save() |

neutron.remove\_ports\_from\_network()

|  |
| --- |
| (ironic/common/neutron.py)  def remove\_ports\_from\_network(task, network\_uuid):  """Deletes the neutron ports created for booting the ramdisk.  :param task: a TaskManager instance.  :param network\_uuid: UUID of a neutron network ports will be deleted from.  :raises: NetworkError  """  # 跟进 mac 地址删除 neutron port  add\_all\_ports = CONF.neutron.add\_all\_ports  if not add\_all\_ports:  macs = [p.address for p in task.ports if p.pxe\_enabled]  else:  macs = [p.address for p in task.ports]  if macs:  params = {  'network\_id': network\_uuid,  'mac\_address': macs,  }  LOG.debug("Removing ports on network %(net)s on node %(node)s.",  {'net': network\_uuid, 'node': task.node.uuid})  remove\_neutron\_ports(task, params) |

#### NeutronNetwork.configure\_tenant\_networks()

|  |
| --- |
| (ironic/drivers/modules/network/neutron.py)  def configure\_tenant\_networks(self, task):  """Configure tenant networks for a node.  :param task: A TaskManager instance.  :raises: NetworkError  """  node = task.node  # 从 task 中获取 ports 信息，实际是从 ironic node 表中获取  ports = task.ports  LOG.info('Mapping instance ports to %s', node.uuid)  if not ports:  msg = \_("No ports are associated with node %s") % node.uuid  LOG.error(msg)  raise exception.NetworkError(msg)  ports = [p for p in ports if not p.portgroup\_id]  portgroups = task.portgroups  client = neutron.get\_client(context=task.context)  pobj\_without\_vif = 0  # 对 node 上 port 和 portgroup 调用 plug\_port\_to\_tenant\_network()  for port\_like\_obj in ports + portgroups:  try:  common.plug\_port\_to\_tenant\_network(task, port\_like\_obj,  client=client)  except exception.VifNotAttached:  pobj\_without\_vif += 1  continue  if pobj\_without\_vif == len(ports + portgroups):  msg = \_("No neutron ports or portgroups are associated with "  "node %s") % node.uuid  LOG.error(msg)  raise exception.NetworkError(msg) |

neutron.plug\_port\_to\_tenant\_network()

|  |
| --- |
| (ironic/drivers/modules/network/common.py)  # 把 ironic 的 port 插到 neutron 租户网络，参数中  # port\_like\_obj 为 ironic 的 port 或 portgroup 对象  def plug\_port\_to\_tenant\_network(task, port\_like\_obj, client=None):  """  """  …  vif\_id = port\_like\_obj.internal\_info.get(TENANT\_VIF\_KEY)  …  if isinstance(port\_like\_obj, objects.Portgroup):  pg\_ports = [p for p in task.ports  if p.portgroup\_id == port\_like\_obj.id]  for port in pg\_ports:  local\_link\_info.append(port.local\_link\_connection)  local\_group\_info = neutron.get\_local\_group\_information(  task, port\_like\_obj)  else:  # We iterate only on ports or portgroups, no need to check  # that it is a port  local\_link\_info.append(port\_like\_obj.local\_link\_connection)  client\_id = port\_like\_obj.extra.get('client-id')  if client\_id:  client\_id\_opt = ({'opt\_name': DHCP\_CLIENT\_ID,  'opt\_value': client\_id})  …  # NOTE(sambetts) Only update required binding: attributes,  # because other port attributes may have been set by the user or  # nova.  port\_attrs = {'binding:vnic\_type': neutron.VNIC\_BAREMETAL,  'binding:host\_id': node.uuid}  # NOTE(kaifeng) Only update mac address when it's available  if port\_like\_obj.address:  port\_attrs['mac\_address'] = port\_like\_obj.address  binding\_profile = {'local\_link\_information': local\_link\_info}  if local\_group\_info:  binding\_profile['local\_group\_information'] = local\_group\_info  port\_attrs['binding:profile'] = binding\_profile  if client\_id\_opt:  port\_attrs['extra\_dhcp\_opts'] = [client\_id\_opt]  is\_smart\_nic = neutron.is\_smartnic\_port(port\_like\_obj)  if is\_smart\_nic:  link\_info = local\_link\_info[0]  LOG.debug('Setting hostname as host\_id in case of Smart NIC, '  'port %(port\_id)s, hostname %(hostname)s',  {'port\_id': vif\_id, 'hostname': link\_info['hostname']})  port\_attrs['binding:host\_id'] = link\_info['hostname']  port\_attrs['binding:vnic\_type'] = neutron.VNIC\_SMARTNIC  if not client:  client = neutron.get\_client(context=task.context)  if is\_smart\_nic:  neutron.wait\_for\_host\_agent(client, port\_attrs['binding:host\_id'])  try:  neutron.update\_neutron\_port(task.context, vif\_id, port\_attrs)  if is\_smart\_nic:  neutron.wait\_for\_port\_status(client, vif\_id, 'ACTIVE')  except openstack\_exc.OpenStackCloudException as e:  msg = (\_('Could not add public network VIF %(vif)s '  'to node %(node)s, possible network issue. %(exc)s') %  {'vif': vif\_id, 'node': node.uuid, 'exc': e})  LOG.error(msg)  raise exception.NetworkError(msg) |

这里面有几个点需要说明：

1. 可以看到，这里面的切换只涉及到 neutron port 的更新，不涉及创建，原因是租户网络的 vif 是 nova 创建的，这是 nova 的统一流程。这是在 nova 中 vm 创建的流程中。
2. Nova 创建实例过程中会调用 ironic 的 vif attach，此时，会给对应关联到 vif 的 ironic port 加上 TENANT\_VIF\_KEY 的信息，因此，这里通过 TENANT\_VIF\_KEY 这个key 去判断 ironic port 是否关联到 vif。
3. 这里更新了 vif 的如下信息：

* ‘binding:vnic\_type’ 更新为 VNIC\_BAREMETAL
* ‘binding:host\_id’ 更新为 ironic node 的 uuid
* mac\_address 更新为对应 ironic port 的 mac 地址
* 'binding:profile' 中增加 'local\_group\_information'

# 裸金属实例的删除流程

## Iroic 处理 deleted 的 provision state

NodeStatesController. \_do\_provision\_action()

|  |
| --- |
| (ironic/api/controllers/v1/node.py)  def \_do\_provision\_action(self, rpc\_node, target, configdrive=None,  clean\_steps=None, deploy\_steps=None,  rescue\_password=None, disable\_ramdisk=None):  …  elif target in (ir\_states.DELETED, ir\_states.UNDEPLOY):  api.request.rpcapi.do\_node\_tear\_down(  api.request.context, rpc\_node.uuid, topic) |

ConductorManager.do\_node\_tear\_down()

|  |
| --- |
| (ironic/conductor/manager.py)  def do\_node\_tear\_down(self, context, node\_id):  """  """  LOG.debug("RPC do\_node\_tear\_down called for node %s.", node\_id)  self.\_concurrent\_action\_limit(action='unprovisioning')  with task\_manager.acquire(context, node\_id, shared=False,  purpose='node tear down') as task:  # Record of any pre-existing agent\_url should be removed.  utils.remove\_agent\_url(task.node)  if task.node.protected:  raise exception.NodeProtected(node=task.node.uuid)  try:  # NOTE(ghe): Valid power driver values are needed to perform  # a tear-down. Deploy info is useful to purge the cache but not  # required for this method.  task.driver.power.validate(task)  except exception.InvalidParameterValue as e:  raise exception.InstanceDeployFailure(\_(  "Failed to validate power driver interface. "  "Can not delete instance. Error: %(msg)s") % {'msg': e})  try:  task.process\_event(  'delete',  callback=self.\_spawn\_worker,  call\_args=(self.\_do\_node\_tear\_down, task,  task.node.provision\_state),  err\_handler=utils.provisioning\_error\_handler)  except exception.InvalidState:  raise exception.InvalidStateRequested(  action='delete', node=task.node.uuid,  state=task.node.provision\_state) |

启动一个异步 worker，调用 \_do\_node\_tear\_down()：

|  |
| --- |
| （ironic/conductor/manager.py）  @task\_manager.require\_exclusive\_lock  def \_do\_node\_tear\_down(self, task, initial\_state):  …  if node.console\_enabled:  notify\_utils.emit\_console\_notification(  task, 'console\_stop', fields.NotificationStatus.START)  try:  # Keep console\_enabled=True for next deployment  task.driver.console.stop\_console(task)  …  # 清理 http 配置，dhcp 配置  task.driver.deploy.clean\_up(task)  # 关机，detach volume  task.driver.deploy.tear\_down(task)  finally:  # NOTE(tenbrae): there is no need to unset conductor\_affinity  # because it is a reference to the most recent conductor which  # deployed a node, and does not limit any future actions.  # But we do need to clear the instance-related fields.  node.instance\_info = {}  node.instance\_uuid = None  utils.wipe\_deploy\_internal\_info(task)  node.del\_driver\_internal\_info('instance')  node.del\_driver\_internal\_info('root\_uuid\_or\_disk\_id')  node.del\_driver\_internal\_info('is\_whole\_disk\_image')  node.del\_driver\_internal\_info('is\_source\_a\_path')  node.del\_driver\_internal\_info('deploy\_boot\_mode')  if node.driver\_internal\_info.get('automatic\_lessee'):  # Remove lessee, as it was automatically added  node.lessee = None  node.del\_driver\_internal\_info('automatic\_lessee')  network.remove\_vifs\_from\_node(task)  node.save()  if node.allocation\_id:  allocation = objects.Allocation.get\_by\_id(task.context,  node.allocation\_id)  allocation.destroy()  # The destroy() call above removes allocation\_id and  # instance\_uuid, refresh the node to get these changes.  node.refresh()  # Begin cleaning  # machine.add\_transition(DELETING, CLEANING, 'clean')  # 从 deleting => cleaning 状态  task.process\_event('clean')  cleaning.do\_node\_clean(task) |

do\_node\_clean()：

|  |
| --- |
| def do\_node\_clean(task, clean\_steps=None, disable\_ramdisk=False):  …  try:  if not disable\_ramdisk:  prepare\_result = task.driver.deploy.prepare\_cleaning(task)  …  try:  conductor\_steps.set\_node\_cleaning\_steps(  task, disable\_ramdisk=disable\_ramdisk)  except Exception as e:  # Catch all exceptions and follow the error handling  # path so things are cleaned up properly.  msg = (\_('Cannot clean node %(node)s: %(msg)s')  % {'node': node.uuid, 'msg': e})  return utils.cleaning\_error\_handler(task, msg)  steps = node.driver\_internal\_info.get('clean\_steps', [])  step\_index = 0 if steps else None  do\_next\_clean\_step(task, step\_index, disable\_ramdisk=disable\_ramdisk) |

# 裸金属的 Inspect 流程

参考：[ironic组件硬件自检服务——ironic-inspector - frankming - 博客园 (cnblogs.com)](https://www.cnblogs.com/frankming/p/15894455.html)

有一个专门的 ironic-inspector 服务，代码在 [openstack/ironic-inspector: Hardware introspection daemon for OpenStack Ironic. Mirror of code maintained at opendev.org. (github.com)](https://github.com/openstack/ironic-inspector)。其提供 / introspection 的api服务，供 ironic 调用。Ironic-inspector 内部也有一个状态机，参考：[ironic-inspector/doc/source/images/states.svg at master · openstack/ironic-inspector · GitHub](https://github.com/openstack/ironic-inspector/blob/master/doc/source/images/states.svg)。

还是在这个 api 中通过状态驱动：

|  |
| --- |
| /v1/nodes/{node\_ident}/states/provision |

|  |
| --- |
| （ironic/api/controllers/v1/node.py）  def \_do\_provision\_action(self, rpc\_node, target, configdrive=None,  clean\_steps=None, deploy\_steps=None,  rescue\_password=None, disable\_ramdisk=None):  …  elif target == ir\_states.VERBS['inspect']:  api.request.rpcapi.inspect\_hardware(  api.request.context, rpc\_node.uuid, topic=topic)  … |

调用 ironic conductor 的 inspect\_hardware() rpc：

|  |
| --- |
| （ironic/conductor/inspection.py）  def inspect\_hardware(task):  …  try:  new\_state = task.driver.inspect.inspect\_hardware(task)  … |

调用 inspect driver 的 inspect\_hardware()。

Ironic-python-agent 对应的 inspect driver 是 Inspector。

## Inspector.inspect\_hardware()

|  |
| --- |
| （ironic/drivers/modules/inspector/interface.py）  class Inspector(base.InspectInterface):  …  else:  # NOTE(dtantsur): spawning a short-living green thread so that  # we can release a lock as soon as possible and allow  # ironic-inspector to operate on the node.  eventlet.spawn\_n(\_start\_inspection, task.node.uuid, task.context)  return states.INSPECTWAIT |

启动一个异步协程调用 \_start\_inspection()。

|  |
| --- |
| （ironic/drivers/modules/inspector/interface.py）  def \_start\_inspection(node\_uuid, context):  """Call to inspector to start inspection."""  try:  client.get\_client(context).start\_introspection(node\_uuid)  … |

这里的client和 start\_introspection() 封装在 openstacksdk 中：

|  |
| --- |
| （openstacksdk/openstack/baremetal\_introspection/v1/\_proxy.py）  def start\_introspection(self, node, manage\_boot=None):  """  """  node = self.\_get\_resource(\_node.Node, node)  res = \_introspect.Introspection.new(  connection=self.\_get\_connection(), id=node.id  )  kwargs = {}  if manage\_boot is not None:  kwargs['manage\_boot'] = manage\_boot  return res.create(self, \*\*kwargs) |

其最终调用的是 ironic-inspector 的 POST / introspection api。

## Ironic-inspector 的 /introspection api

|  |
| --- |
| （ironic-inspector/ironic\_inspector/main.py）  @api('/v1/introspection/<node\_id>',  rule="introspection:{}",  verb\_to\_rule\_map={'GET': 'status', 'POST': 'start'},  methods=['GET', 'POST'])  def api\_introspection(node\_id):  if flask.request.method == 'POST':  args = flask.request.args  manage\_boot = args.get('manage\_boot', 'True')  try:  manage\_boot = strutils.bool\_from\_string(manage\_boot, strict=True)  except ValueError:  raise utils.Error(\_('Invalid boolean value for manage\_boot: %s') %  manage\_boot, code=400)  if manage\_boot and not CONF.can\_manage\_boot:  raise utils.Error(\_('Managed boot is requested, but this '  'installation cannot manage boot ('  '(can\_manage\_boot set to False)'),  code=400)  client = get\_client\_compat()  client.call({}, 'do\_introspection', node\_id=node\_id,  manage\_boot=manage\_boot,  token=flask.request.headers.get('X-Auth-Token'))  return \_generate\_empty\_response(202)  else:  node\_info = node\_cache.get\_node(node\_id)  return flask.json.jsonify(generate\_introspection\_status(node\_info)) |

Ironic-inspector conductor 的 do\_introspection()：

|  |
| --- |
| （ironic-inspector/ironic\_inspector/conductor/manager.py）  @messaging.expected\_exceptions(utils.Error)  def do\_introspection(self, context, node\_id, token=None,  manage\_boot=True):  introspect.introspect(node\_id, token=token, manage\_boot=manage\_boot) |

|  |
| --- |
| (ironic-inspector/ironic\_inspector/introspect.py) |

## Ironic-python-agent 回调

Ironic 会把 ipa-inspection-callback-url 注入到 ipa 中，ipa 会在 inspect() 过程中回调这个 url，具体是 IronicInspection.\_run() -> inspector.inspect() -> call\_inspector()：

|  |
| --- |
| （ironic-python-agent/ironic\_python\_agent/inspector.py）  def call\_inspector(data, failures):  """Post data to inspector."""  data['error'] = failures.get\_error()  LOG.info('posting collected data to %s', CONF.inspection\_callback\_url)  LOG.debug('collected data: %s',  {k: v for k, v in data.items() if k not in \_NO\_LOGGING\_FIELDS})  encoder = encoding.RESTJSONEncoder()  data = encoder.encode(data)  verify, cert = utils.get\_ssl\_client\_options(CONF)  @tenacity.retry(  retry=tenacity.retry\_if\_exception\_type(  requests.exceptions.ConnectionError),  stop=tenacity.stop\_after\_attempt(\_RETRY\_ATTEMPTS),  wait=tenacity.wait\_fixed(\_RETRY\_WAIT),  reraise=True)  def \_post\_to\_inspector():  return requests.post(CONF.inspection\_callback\_url, data=data,  verify=verify, cert=cert)  resp = \_post\_to\_inspector()  if resp.status\_code >= 400:  LOG.error('inspector %s error %d: %s, proceeding with lookup',  CONF.inspection\_callback\_url,  resp.status\_code, resp.content.decode('utf-8'))  return  return resp.json() |

# Ironic 中的存储

Ironic 中存在一个存储相关的 Interface：StorageInterface。Ironic 明明是托管的实体的服务器，为什么会涉及到存储呢？这涉及一个功能：从远端存储启动（boot from a remote volume）。参考：[Boot From Volume — ironic 21.5.0.dev117 documentation (openstack.org)](https://docs.openstack.org/ironic/latest/admin/boot-from-volume.html)。

## Boot From Volume 原理

Boot From Volume 是物理服务器本身支持的功能，叫做 iscsi boot，以 ubuntu 为例，需要几个东西配合工作：

* Ipxe 支持，在网卡上配置 sanhook，参考 [iPXE - open source boot firmware [cmd:sanhook]](https://ipxe.org/cmd/sanhook)，其中指定 iscsi-target 的参数：

sanhook --drive 0x80 iscsi:10.0.4.1:::1:iqn.2010-04.org.ipxe.dolphin:storage

其实 ipxe 支持的不仅仅是 iscsi，而是普通的 http 地址就可以。Ironic 中 cinder 就是一个 http 地址。

* 镜像中需要做配置，如果使用 disk-image-create 工具制作镜像，需要带上 iscsi-boot 参数，以 ubuntu 为例，具体会做以下操作：
  + 安装 open-iscsi
  + Creates the ``etc/iscsi/iscsi.initramfs`` configuration file and sets ``ISCSI\_AUTO=true`` within it.

下面结合 Ironic 的代码查看这个流程。

### 1 在node 上添加 volume 信息

可以参考 /v1/volume/targets 这个 api，[Bare Metal API — Ironic API Reference documentation (openstack.org)](https://docs.openstack.org/api-ref/baremetal/?expanded=create-volume-target-detail,list-volume-targets-detail)

在 VolumeTargetsController.post() 中：

|  |
| --- |
| (ironic/api/controllers/v1/volume\_target.py)  @METRICS.timer('VolumeTargetsController.post')  @method.expose(status\_code=http\_client.CREATED)  @method.body('target')  @args.validate(target=TARGET\_VALIDATOR)  def post(self, target): |

### 2 配置 ipxe

|  |
| --- |
| (ironic/common/pxe\_utils.py)  def get\_volume\_pxe\_options(task): |

中配置 ipxe 配置。

## StorageInterface

### attach\_volumes()

|  |
| --- |
| (ironic/drivers/base.py)  @abc.abstractmethod  def attach\_volumes(self, task):  """Informs the storage subsystem to attach all volumes for the node.  :param task: A TaskManager instance.  :raises: UnsupportedDriverExtension  """ |

对于 cinder 来说，attach\_volumes() 就是调用 cinder 的 api，把 volume 设置为 attached 状态，并且指定绑定者为 ironic。

### should\_write\_image()

|  |
| --- |
| (ironic/drivers/base.py)  @abc.abstractmethod  def should\_write\_image(self, task):  """Determines if deploy should perform the image write-out.  :param task: A TaskManager instance.  :returns: Boolean value to indicate if the interface expects  the image to be written by Ironic.  :raises: UnsupportedDriverExtension  """ |

判断部署时是否需要执行 write image。

其主要在 write\_image() 时调用，如果返回 false，就不会执行 write image 操作了：

|  |
| --- |
| （ironic/drivers/modules/agent.py）  def write\_image(self, task):  if not task.driver.storage.should\_write\_image(task):  return  … |

# metal3-dev-env 中的示例解析

metal3-dev-env 中用 libvirt 的虚机模拟了两个裸金属节点，这里解析一下其在 metal3 中的逻辑。

关键在于 03\_launch\_mgmt\_cluster.sh 这个脚本中创建 BareMetalHost：

|  |
| --- |
| （03\_launch\_mgmt\_cluster.sh）  #  # Apply the BMH CRs  #  function apply\_bm\_hosts() {  NAMESPACE=$1  pushd "${BMOPATH}"  list\_nodes | make\_bm\_hosts > "${WORKING\_DIR}/bmhosts\_crs.yaml"  if [[ -n "$(list\_nodes)" ]]; then  echo "bmhosts\_crs.yaml is applying"  while ! kubectl apply -f "${WORKING\_DIR}/bmhosts\_crs.yaml" -n "$NAMESPACE" &>/dev/null; do  sleep 3  done  echo "bmhosts\_crs.yaml is successfully applied"  fi  popd  } |

这个文件的内容是：

|  |
| --- |
| （/opt/metal3-dev-env/bmhosts\_crs.yaml）  ---  apiVersion: v1  kind: Secret  metadata:  name: node-0-bmc-secret  type: Opaque  data:  username: YWRtaW4=  password: cGFzc3dvcmQ=  ---  apiVersion: metal3.io/v1alpha1  kind: BareMetalHost  metadata:  name: node-0  spec:  online: true  bootMACAddress: 00:2a:ec:ab:5f:11  bootMode: legacy  bmc:  address: ipmi://192.168.111.1:6230  credentialsName: node-0-bmc-secret  ---  apiVersion: v1  kind: Secret  metadata:  name: node-1-bmc-secret  type: Opaque  data:  username: YWRtaW4=  password: cGFzc3dvcmQ=  ---  apiVersion: metal3.io/v1alpha1  kind: BareMetalHost  metadata:  name: node-1  spec:  online: true  bootMACAddress: 00:2a:ec:ab:5f:15  bootMode: legacy  bmc:  address: redfish+http://192.168.111.1:8000/redfish/v1/Systems/57b0a66e-5ec2-4cff-9fe5-361c84a191a7  credentialsName: node-1-bmc-secret |

可以看到，其中定义了 node-0 和 node-1 两个 BareMetalHost，它们的 bmc 接口密码通过两个 Secret 对象来定义。

node-0 和 node-1 是 libvirt 模拟好的，其对应的 bmc 接口通过 vbmc 模拟。

我们查看两个 BareMetalHost 当前的状态：



可以看到，两个 BareMetalHost 的当前状态是 available，即已经完成了 Inspecting 和 Preparing。那么问题来了，我们在 yaml 描述中只指定了 online 为 true，这个状态流转的逻辑是什么？自动进入 available？

这个状态字段在 BareMetalHostStatus 中的 ProvisionStatus 中，可以参考 BareMetalHostStatus 的说明。

状态的流转在 HostStateMachine 这个fsm 状态机中，初始状态在 newHostStateMachine() 时指定：

|  |
| --- |
| （controllers/metal3.io/baremetalhost\_controller.go）  func (r \*BareMetalHostReconciler) Reconcile(ctx context.Context, request ctrl.Request) (result ctrl.Result, err error) {  …  initialState := host.Status.Provisioning.State  …  stateMachine := newHostStateMachine(host, r, prov, haveCreds)  …  } |

初始状态应该是 StateNone。进入 handleNone() 处理：

|  |
| --- |
| （controllers/metal3.io/host\_state\_machine.go）  func (hsm \*hostStateMachine) handleNone(info \*reconcileInfo) actionResult {  // No state is set, so immediately move to either Registering or Unmanaged  if hsm.Host.HasBMCDetails() {  hsm.NextState = metal3v1alpha1.StateRegistering  } else {  …  } |

我们设置了BMC 信息，因此，进入 StateRegistering，由 handleRegistering() 处理：

|  |
| --- |
| （controllers/metal3.io/host\_state\_machine.go）  func (hsm \*hostStateMachine) handleRegistering(info \*reconcileInfo) actionResult {  // Getting to the state handler at all means we have successfully  // registered using the current BMC credentials, so we can move to the  // next state. We will not return to the Registering state, even  // if the credentials change and the Host must be re-registered.  if hsm.Host.Spec.ExternallyProvisioned {  hsm.NextState = metal3v1alpha1.StateExternallyProvisioned  } else if inspectionDisabled(hsm.Host) {  hsm.NextState = metal3v1alpha1.StatePreparing  } else {  hsm.NextState = metal3v1alpha1.StateInspecting  }  hsm.Host.Status.ErrorCount = 0  return actionComplete{}  } |

进入 StateInspecting，由 handleInspecting() 处理：

|  |
| --- |
| （controllers/metal3.io/host\_state\_machine.go）  func (hsm \*hostStateMachine) handleInspecting(info \*reconcileInfo) actionResult {  actResult := hsm.Reconciler.actionInspecting(hsm.Provisioner, info)  if \_, complete := actResult.(actionComplete); complete {  hsm.NextState = metal3v1alpha1.StatePreparing  hsm.Host.Status.ErrorCount = 0  }  return actResult  } |

在 Reconciler.actionInspecting() 中会调用 ironic 的 inpect 逻辑。

Inspect 完成之后，进入 StatePreparing。由 handlePreparing() 处理：

|  |
| --- |
| （controllers/metal3.io/host\_state\_machine.go）  func (hsm \*hostStateMachine) handlePreparing(info \*reconcileInfo) actionResult {  actResult := hsm.Reconciler.actionPreparing(hsm.Provisioner, info)  if \_, complete := actResult.(actionComplete); complete {  hsm.Host.Status.ErrorCount = 0  hsm.NextState = metal3v1alpha1.StateAvailable  }  return actResult  } |

在 Reconciler. actionPreparing() 中完成后进入 StateAvailable。由 handleAvailable() 处理：

|  |
| --- |
| （controllers/metal3.io/host\_state\_machine.go）  func (hsm \*hostStateMachine) handleAvailable(info \*reconcileInfo) actionResult {  …  // ErrorCount is cleared when appropriate inside actionManageAvailable  actResult := hsm.Reconciler.actionManageAvailable(hsm.Provisioner, info)  if \_, complete := actResult.(actionComplete); complete {  hsm.NextState = metal3v1alpha1.StateProvisioning  }  return actResult  } |

|  |
| --- |
| (controllers/metal3.io/baremetalhost\_controller.go)  func (r \*BareMetalHostReconciler) actionManageAvailable(prov provisioner.Provisioner, info \*reconcileInfo) actionResult {  if info.host.NeedsProvisioning() {  clearError(info.host)  return actionComplete{}  }  return r.manageHostPower(prov, info)  } |

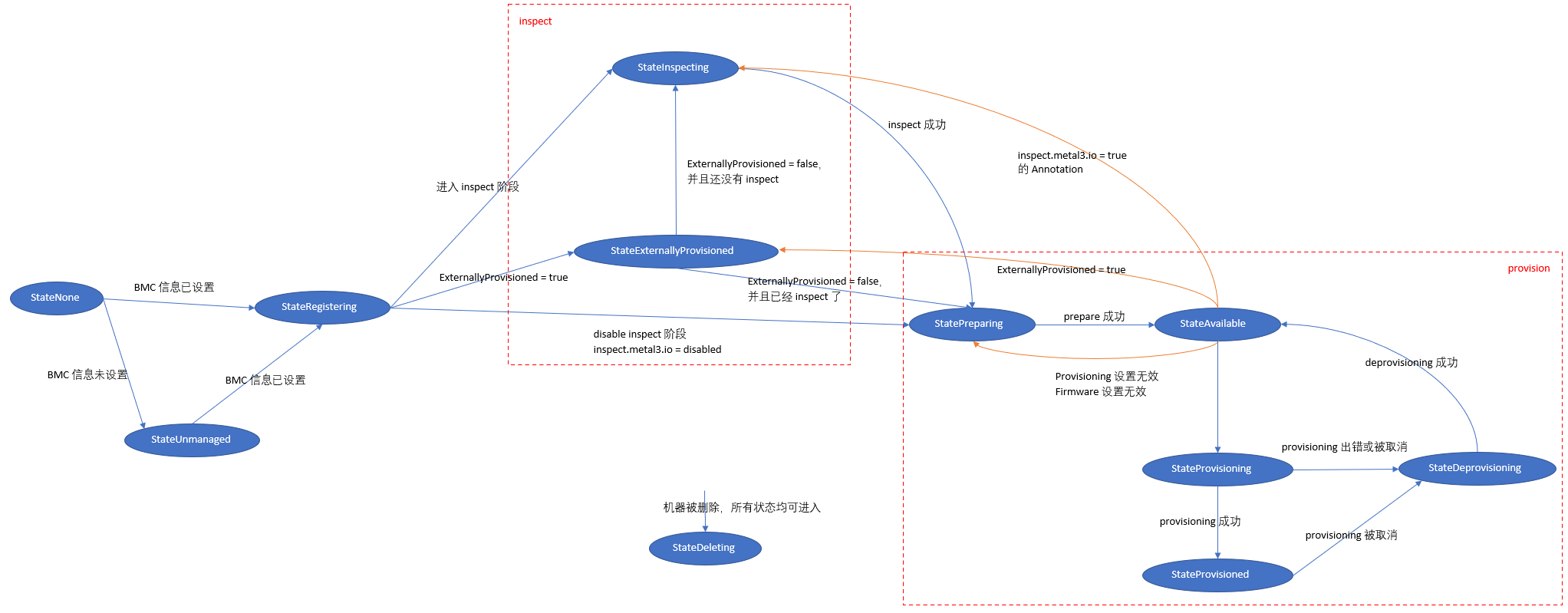
可以看到，在 StateAvailable 阶段要不要进入 StateProvisioning，取决于 NeedsProvisioning() 的结果：

|  |
| --- |
| （apis/metal3.io/v1alpha1/baremetalhost\_types.go）  func (host \*BareMetalHost) NeedsProvisioning() bool {  if !host.Spec.Online {  // The host is not supposed to be powered on.  return false  }  return host.hasNewImage() || host.hasNewCustomDeploy()  } |

条件是：

* host.Spec.Online 要为 true
* 必须要设置了 image host.Spec.Image 或 host.Spec.CustomDeploy。

# ProvisioningState 状态机



各状态说明：

* StateNone：初始状态
* StateUnmanaged（稳态）：BareMetalHost 没有设置 BMC 信息，进入 StateNone，目前只会从 StateNone -> StateUnmanaged。StateUnmanaged 状态下会持续检查 BMC 信息，如果设置了，进入 StateRegistering，否则留在 StateUnmanaged
* StateRegistering：一个中间状态，根据条件进入以下三个状态：
  + StateExternallyProvisioned：Host.Spec.ExternallyProvisioned 为 true
  + StatePreparing：通过 inspect.metal3.io = disabled 的 annotation 跳过 inspect 阶段
  + StateInspecting：正常进入 Inspecting 阶段
* StateExternallyProvisioned（稳态）：根据条件进入以下状态：
  + Host.Spec.ExternallyProvisioned为true时，其本身是一个稳态，如果调用 provisioner.Adopt() 成功（例如 ironic 进入 active 状态），就会保持在这个状态。表示由外部的系统 Provision，区别于 StateProvisioned（内部 Provision）
  + StateInspecting：Host.Spec.ExternallyProvisioned = false 时，并且需要 inspect，进入 StateInspecting
  + StatePreparing：Host.Spec.ExternallyProvisioned = false 并且不需要 inspect 时，进入 StatePreparing
* StateInspecting：非稳态，inspect 过程中。调用 provisioner.InspectHardware() 收集硬件信息。

# metal3 中的 CRD 对象和 Controller

## BareMetalHost

表示一台裸金属机器

## PreprovisioningImage

参考：[metal3-docs/image-builder-integration.md at main · metal3-io/metal3-docs · GitHub](https://github.com/metal3-io/metal3-docs/blob/main/design/baremetal-operator/image-builder-integration.md)

如文档中所述，增加这个对象的目的是让用户能提供自己的包含 ramdisk image（对应 ironic 中包含 ironic-python-agent 的）。对应有一个控制器 PreprovisioningImageReconciler。

PreprovisioningImage 的基本工作流程是：

* baremetalhost controller 检查到需要自定义 ramdisk image，创建一个 PreprovisioningImage 对象
* preprovisioningimage controller 检查到 PreprovisioningImage 对象，调用 imageprovider 准备 image。

下面是各个步骤的详细流程。

### PreprovisioningImage 创建

在 StateRegistering 阶段，registerHost() 函数中：

|  |
| --- |
| （controllers/metal3.io/baremetalhost\_controller.go）  func (r \*BareMetalHostReconciler) registerHost(prov provisioner.Provisioner, info \*reconcileInfo) actionResult {  …  // 调用 provisioner.PreprovisioningImageFormats() 获取 provisioner支持的PreprovisioningImage 的格式，如果不需要 PreprovisioningImage，返回 nil  preprovImgFormats, err := prov. PreprovisioningImageFormats ()  if err != nil {  return actionError{err}  }  …  // 创建 PreprovisioningImage 对象  preprovImg, err := r.getPreprovImage(info, preprovImgFormats)  if err != nil {  if errors.As(err, &imageBuildError{}) {  return recordActionFailure(info, metal3v1alpha1.RegistrationError, err.Error())  }  return actionError{err}  }  // 把裸金属注册到 provisioner，例如对于 ironic 就是创建 node  // 并且验证裸金属的访问和用户名密码等  provResult, provID, err := prov.ValidateManagementAccess(  provisioner.ManagementAccessData{  BootMode: info.host.Status.Provisioning.BootMode,  AutomatedCleaningMode: info.host.Spec.AutomatedCleaningMode,  State: info.host.Status.Provisioning.State,  CurrentImage: getCurrentImage(info.host),  PreprovisioningImage: preprovImg,  HasCustomDeploy: hasCustomDeploy(info.host),  },  credsChanged,  info.host.Status.ErrorType == metal3v1alpha1.RegistrationError)  // 如果 provisioner 要求 PreprovisioningImage，但是没有提供，则报错，如果 PreprovisioningImage 还没准备好，则等待  if errors.Is(err, provisioner.ErrNeedsPreprovisioningImage) &&  preprovImgFormats != nil {  if preprovImg == nil {  // PreprovisioningImage 还没准备好，等待  waitingForPreprovImage.Inc()  return actionContinue{preprovImageRetryDelay}  }  return recordActionFailure(info, metal3v1alpha1.RegistrationError,  "Preprovisioning Image is not acceptable to provisioner")  }  …  } |

### preprovisioningimage controller 处理

|  |
| --- |
| (controllers/metal3.io/preprovisioningimage\_controller.go)  func (r \*PreprovisioningImageReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {  …  // update() 中可能会更新 img，如果更新了，返回值 changed = true  changed, err := r.update(&img, log)  if k8serrors.IsNotFound(err) {  delay := getErrorRetryDelay(img.Status)  log.Info("requeuing to check for secret", "after", delay)  result.RequeueAfter = delay  }  notReady := imageprovider.ImageNotReady{}  if errors.As(err, &notReady) {  log.Info("image is not ready yet, requeuing", "after", minRetryDelay)  if setUnready(img.GetGeneration(), &img.Status, err.Error()) {  changed = true  }  result.RequeueAfter = minRetryDelay  }  if changed {  log.Info("updating status")  err = r.Status().Update(ctx, &img)  }  return result, err  } |

update() 函数：

|  |
| --- |
| (controllers/metal3.io/preprovisioningimage\_controller.go)  func (r \*PreprovisioningImageReconciler) update(img \*metal3.PreprovisioningImage, log logr.Logger) (bool, error) {  generation := img.GetGeneration()  // 检查 ImageProvider 是否支持对应架构的 image  if !r.ImageProvider.SupportsArchitecture(img.Spec.Architecture) {  log.Info("image architecture not supported", "architecture", img.Spec.Architecture)  return setError(generation, &img.Status, reasonImageConfigurationError, "Architecture not supported"), nil  }  // 从 PreprovisioningImage.Spec 中可接受的 image format 中选择一个  format := r.getImageFormat(img.Spec, log)  if format == "" {  return setError(generation, &img.Status, reasonImageConfigurationError, "No acceptable image format supported"), nil  }  // 根据 PreprovisioningImage.Spec.NetworkDataName 查找相应的 Secret  secretManager := secretutils.NewSecretManager(log, r.Client, r.APIReader)  networkData, secretStatus, err := getNetworkData(secretManager, img)  if err != nil {  if k8serrors.IsNotFound(err) {  log.Info("network data Secret does not exist")  return setError(generation, &img.Status, reasonImageMissingNetworkData, "NetworkData secret not found"), err  }  return false, err  }  // 检查 PreprovisioningImage.Spec 和 PreprovisioningImage.Status 判断配置是否发生了变化  if configChanged(img, format, secretStatus) {  reason := "Config changed"  if meta.IsStatusConditionTrue(img.Status.Conditions, string(metal3.ConditionImageReady)) {  // 如果当前状态是 Ready，把状态从 Ready 变成 Unready  // Ensure we mark the status as not ready before we remove the build  // from the image cache.  setUnready(generation, &img.Status, reason)  } else {  // 如果当前状态不是 Ready，先把当前的 Image 抛弃  if err := r.discardExistingImage(img, log); err != nil {  return false, err  }  // Set up all the data before building the image and adding the URL,  // so that even if we fail to write the built image status and the  // config subsequently changes, the image cache cannot leak.  // 先把新Image 信息保存到 Status 中  setImage(generation, &img.Status, imageprovider.GeneratedImage{},  format, secretStatus, img.Spec.Architecture,  reason)  }  return true, nil  }  var networkDataContent imageprovider.NetworkData  if networkData != nil {  networkDataContent = networkData.Data  }  // 调用 ImageProvider 准备 Image  image, err := r.ImageProvider.BuildImage(imageprovider.ImageData{  ImageMetadata: img.ObjectMeta.DeepCopy(),  Format: format,  Architecture: img.Spec.Architecture,  NetworkDataStatus: secretStatus,  }, networkDataContent, log)  if err != nil {  failure := imageprovider.ImageBuildInvalid{}  if errors.As(err, &failure) {  log.Info("image build failed", "error", "err")  return setError(generation, &img.Status, reasonImageBuildInvalid, failure.Error()), nil  }  return false, err  }  log.Info("image URL available", "url", image, "format", format)  // 保存 Image 信息到 Status  return setImage(generation, &img.Status, image, format,  secretStatus, img.Spec.Architecture,  "Generated image"), nil  } |

## ironicProvisioner 中的处理

## ImageProvider 接口

从 preprovisioningimage controller 中可以看到，PreprovisioningImage

# Provisioner 接口说明

Provisioner 接口定义为：

|  |
| --- |
| （pkg/provisioner/provisioner.go）  type Provisioner interface {  } |

## Metal3 能支持多个 provisioner 同时运行吗？

目前不能，provisioner 初始化在：

|  |
| --- |
| func main() {  …  var provisionerFactory provisioner.Factory  if runInTestMode {  ctrl.Log.Info("using test provisioner")  provisionerFactory = &fixture.Fixture{}  } else if runInDemoMode {  ctrl.Log.Info("using demo provisioner")  provisionerFactory = &demo.Demo{}  } else {  provLog := zap.New(zap.UseFlagOptions(&logOpts)).WithName("provisioner")  provisionerFactory = ironic.NewProvisionerFactory(provLog, preprovImgEnable)  }  …  } |

## ValidateManagementAccess()

在 StateRegistering 阶段调用，验证裸金属配置的访问信息，并注册到 ironic 中（创建 Node）。参考 PreprovisioningImage 中的说明

## PreprovisioningImageFormats()

返回 provisioner 支持的 PreprovisioningImage 格式，参考 PreprovisioningImage 中的说明。

## Adopt()

这个接口用于 StateExternallyProvisioned 状态，用于通知 provisioner 这个host 被 externally-provisioned。

|  |
| --- |
| （pkg/provisioner/provisioner.go）  type Provisioner interface {  // Adopt brings an externally-provisioned host under management by  // the provisioner.  Adopt(data AdoptData, restartOnFailure bool) (result Result, err error)  } |

以 ironic 为例，参考 ironic 的状态机，manageable 到 active 的正常流程是 manageable 到 available，然后经过部署（deploying）到 active。通过 adopt 操作，可以直接从 manageable 到 active，跳过 available 和 deploying。

（参考 [Node adoption — ironic 21.5.0.dev115 documentation (openstack.org)](https://docs.openstack.org/ironic/latest/admin/adoption.html)）

Ironic adopt 的实现在 ConductorManager. \_do\_adoption()：

|  |
| --- |
| （ironic/conductor/manager.py）  def \_do\_adoption(self, task):  try:  utils.update\_image\_type(task.context, task.node)  if deploy\_utils.get\_boot\_option(node) != 'local':  # 验证 boot 配置  task.driver.boot.validate(task)  # NOTE(TheJulia): While task.driver.boot.validate() is called  # above, and task.driver.power.validate() could be called, it  # is called as part of the transition from ENROLL to MANAGEABLE  # states. As such it is redundant to call here.  self.\_do\_takeover(task)  LOG.info("Successfully adopted node %(node)s",  {'node': node.uuid})  task.process\_event('done') |

\_do\_takeover()：

|  |
| --- |
| （ironic/conductor/manager.py）  def \_do\_takeover(self, task):  """Take over this node.  Prepares a node for takeover by this conductor, performs the takeover,  and changes the conductor associated with the node. The node with the  new conductor affiliation is saved to the DB.  :param task: a TaskManager instance  """  LOG.debug('Conductor %(cdr)s taking over node %(node)s',  {'cdr': self.host, 'node': task.node.uuid})  task.driver.deploy.prepare(task)  task.driver.deploy.take\_over(task)  … |

其中调用 deploy interface 的 take\_over() 接口。

这里再看下 metal3 中 ironic provisioner 的 Adopt() 实现：

|  |
| --- |
| （pkg/provisioner/ironic/ironic.go）  // Adopt notifies the provisioner that the state machine believes the host  // to be currently provisioned, and that it should be managed as such.  func (p \*ironicProvisioner) Adopt(data provisioner.AdoptData, restartOnFailure bool) (result provisioner.Result, err error) {  ironicNode, err := p.getNode()  if err != nil {  return transientError(err)  }  switch nodes.ProvisionState(ironicNode.ProvisionState) {  case nodes.Enroll, nodes.Verifying:  return transientError(fmt.Errorf("Invalid state for adopt: %s",  ironicNode.ProvisionState))  case nodes.Manageable:  \_, hasImageSource := ironicNode.InstanceInfo["image\_source"]  \_, hasBootISO := ironicNode.InstanceInfo["boot\_iso"]  if data.State == metal3v1alpha1.StateDeprovisioning &&  !(hasImageSource || hasBootISO) {  // If we got here after a fresh registration and image data is  // available, it should have been added to the node during  // registration. If it isn't present then we got here due to a  // failed cleaning on deprovision. The node will be cleaned again  // before the next provisioning, so just allow the controller to  // continue without adopting.  p.log.Info("no image info; not adopting", "state", ironicNode.ProvisionState)  return operationComplete()  }  // 进入 adopting  return p.changeNodeProvisionState(  ironicNode,  nodes.ProvisionStateOpts{  Target: nodes.TargetAdopt,  },  )  case nodes.Adopting:  // adopting 过程中，继续等待  return operationContinuing(provisionRequeueDelay)  case nodes.AdoptFail:  // adopt 失败，重试或返回失败  if restartOnFailure {  return p.changeNodeProvisionState(  ironicNode,  nodes.ProvisionStateOpts{  Target: nodes.TargetAdopt,  },  )  }  return operationFailed(fmt.Sprintf("Host adoption failed: %s",  ironicNode.LastError))  case nodes.Active:  // adopt 成功  // Empty Fault means that maintenance was set manually, not by Ironic  if ironicNode.Maintenance && ironicNode.Fault == "" && data.State != metal3v1alpha1.StateDeleting {  p.log.Info("active node was found to be in maintenance, updating", "state", data.State)  return p.setMaintenanceFlag(ironicNode, false, "")  }  default:  }  return operationComplete()  } |

## InspectHardware()

Inspect 操作，在 StateInspecting 状态调用：

|  |
| --- |
| （pkg/provisioner/provisioner.go）  type Provisioner interface {  // Adopt brings an externally-provisioned host under management by  // the provisioner.  // 参数中，InspectData 用于 meta3 向 provisioner 传递 inspect 的信息，如bootMode  // restartOnFailure 表示 InspectFail 后要不要重试  // refresh 表示是否要重新进入 inspect  // forceReboot 表示是否要终止 inspecting 进入重启  // 返回值中，started 表示 inspect 是否开始了  // details 返回 inspect 获取到的硬件信息  InspectHardware(data InspectData, restartOnFailure, refresh, forceReboot bool) (result Result, started bool, details \*metal3v1alpha1.HardwareDetails, err error)  } |

## UpdateHardwareState()

Meta3 从 provisioner 查询当前的硬件状态信息，如电源状态等。

## Prepare()

在 StatePreparing 阶段调用的接口，实际对应的是 ironic 的 clean，即对当前的信息进行清理，准备应用新的配置。

|  |
| --- |
| （pkg/provisioner/provisioner.go）  // Prepare remove existing configuration and set new configuration  // 参数中，  Prepare(data PrepareData, unprepared bool, restartOnFailure bool) (result Result, started bool, err error) |

## Provision()

在 StateProvisioning 阶段调用，执行裸金属的部署。

## Deprovision()

在 StateDeprovisioning 阶段调用，回滚部署

## Delete()

在 StateDeleting 阶段调用，从 provisioner 中删除，即从 ironic 中删除 node。

# HardwareData

参考 InspectHardware() 和 docs/api.md 文档。

# BareMetalHost 对象的字段说明

BareMetalHost 定义在：

|  |
| --- |
| （apis/metal3.io/v1alpha1/baremetalhost\_types.go）  type BareMetalHost struct {  metav1.TypeMeta `json:",inline"`  metav1.ObjectMeta `json:"metadata,omitempty"`  Spec BareMetalHostSpec `json:"spec,omitempty"`  Status BareMetalHostStatus `json:"status,omitempty"`  } |

## BareMetalHostSpec

BareMetalHostSpec 是用户指定的部分，下面看下其中的字段。

### Taints

|  |
| --- |
| Taints []corev1.Taint `json:"taints,omitempty"` |

### BMCDetails

保存裸金属带外管理的地址信息。

|  |
| --- |
| BMC BMCDetails `json:"bmc,omitempty"` |

BMCDetails 的定义：

|  |
| --- |
| （apis/metal3.io/v1alpha1/baremetalhost\_types.go）  // BMCDetails contains the information necessary to communicate with  // the bare metal controller module on host.  type BMCDetails struct {  // Address holds the URL for accessing the controller on the  // network.  // Url 地址  Address string `json:"address"`  // The name of the secret containing the BMC credentials (requires  // keys "username" and "password").  // 访问权限信息，如用户名密码等，并没有直接写在这，而是通过 k8s 的 Secret  // 来传入，然后在这里指定 Secret 对象的名称（注意，这个 secret 对象要在 metal3 的 namespace 中）  CredentialsName string `json:"credentialsName"`  // DisableCertificateVerification disables verification of server  // certificates when using HTTPS to connect to the BMC. This is  // required when the server certificate is self-signed, but is  // insecure because it allows a man-in-the-middle to intercept the  // connection.  DisableCertificateVerification bool `json:"disableCertificateVerification,omitempty"`  } |

下面是一个示例：

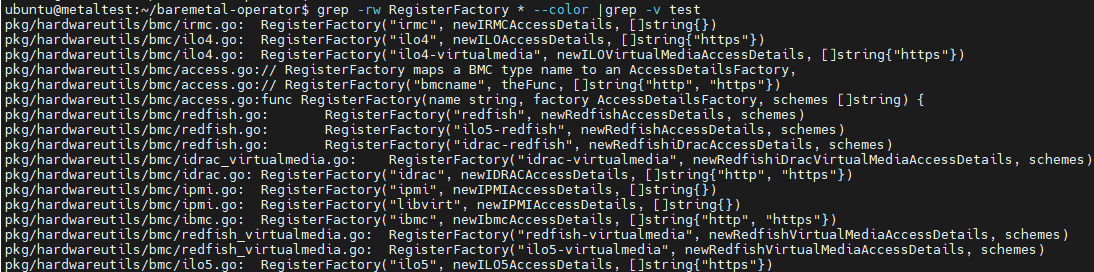
|  |
| --- |
| bmc:  address: ipmi://192.168.111.1:6230  credentialsName: node-0-bmc-secret |

对应的 secret 对象信息：

|  |
| --- |
| ubuntu@metaltest:~/baremetal-operator$ kubectl get secrets node-0-bmc-secret -n metal3 -o yaml  apiVersion: v1  data:  password: cGFzc3dvcmQ=  username: YWRtaW4=  kind: Secret  metadata:  annotations:  kubectl.kubernetes.io/last-applied-configuration: |  {"apiVersion":"v1","data":{"password":"cGFzc3dvcmQ=","username":"YWRtaW4="},"kind":"Secret","metadata":{"annotations":{},"name":"node-0-bmc-secret","namespace":"metal3"},"type":"Opaque"}  creationTimestamp: "2023-05-26T06:35:41Z"  finalizers:  - baremetalhost.metal3.io/secret  labels:  environment.metal3.io: baremetal  name: node-0-bmc-secret  namespace: metal3  ownerReferences:  - apiVersion: metal3.io/v1alpha1  kind: BareMetalHost  name: node-0  uid: 51c2c940-03a3-4dc6-9927-6758ceb48209  resourceVersion: "2666"  uid: d5ea7535-62f9-4a93-8feb-c48ca627c265  type: Opaque |

#### Bmc 地址格式

关于 bmc 地址格式，前面的示例中是：ipmi://192.168.111.1:6230，其中 impi为支持的带外管理实现，在源码中称为 factory。这些factory通过 RegisterFactory() 进行注册：



可以看到支持的有 ipmi，ilo4，rdrac，libvirt 等。

|  |
| --- |
| (pkg/hardwareutils/bmc/access.go)  func RegisterFactory(name string, factory AccessDetailsFactory, schemes []string) {  factories[name] = factory  for \_, scheme := range schemes {  factories[fmt.Sprintf("%s+%s", name, scheme)] = factory  }  } |

从 RegisterFactory() 中可以看到，factory name 为支持的带外管理机制，如 impi，redfish 等，scheme 为通信协议，如 http、https。所以，地址格式的前缀为 factory name + scheme 格式，如：

|  |
| --- |
| redfish+http://192.168.111.1:8000/redfish/v1/Systems/57b0a66e-5ec2-4cff-9fe5-361c84a191a7 |

通过这个前缀来查找 factory 的实现。

### RAIDConfig

指定裸金属的 raid 配置。

|  |
| --- |
| RAID \*RAIDConfig `json:"raid,omitempty"` |

RAIDConfig 定义为：

|  |
| --- |
| （apis/metal3.io/v1alpha1/baremetalhost\_types.go）  // RAIDConfig contains the configuration that are required to config RAID in Bare Metal server  type RAIDConfig struct {  // The list of logical disks for hardware RAID, if rootDeviceHints isn't used, first volume is root volume.  // You can set the value of this field to `[]` to clear all the hardware RAID configurations.  // +optional  // +nullable  HardwareRAIDVolumes []HardwareRAIDVolume `json:"hardwareRAIDVolumes"`  // The list of logical disks for software RAID, if rootDeviceHints isn't used, first volume is root volume.  // If HardwareRAIDVolumes is set this item will be invalid.  // The number of created Software RAID devices must be 1 or 2.  // If there is only one Software RAID device, it has to be a RAID-1.  // If there are two, the first one has to be a RAID-1, while the RAID level for the second one can be 0, 1, or 1+0.  // As the first RAID device will be the deployment device,  // enforcing a RAID-1 reduces the risk of ending up with a non-booting node in case of a disk failure.  // Software RAID will always be deleted.  // +kubebuilder:validation:MaxItems=2  // +optional  // +nullable  SoftwareRAIDVolumes []SoftwareRAIDVolume `json:"softwareRAIDVolumes"`  } |

### FirmwareConfig

指定裸金属 bios 固件的配置。

|  |
| --- |
| Firmware \*FirmwareConfig `json:"firmware,omitempty"` |

FirmwareConfig 定义为：

|  |
| --- |
| （apis/metal3.io/v1alpha1/baremetalhost\_types.go）  // FirmwareConfig contains the configuration that you want to configure BIOS settings in Bare metal server  type FirmwareConfig struct {  // Supports the virtualization of platform hardware.  // This supports following options: true, false.  // +kubebuilder:validation:Enum=true;false  // 开启虚拟化  VirtualizationEnabled \*bool `json:"virtualizationEnabled,omitempty"`  // Allows a single physical processor core to appear as several logical processors.  // This supports following options: true, false.  // +kubebuilder:validation:Enum=true;false  SimultaneousMultithreadingEnabled \*bool `json:"simultaneousMultithreadingEnabled,omitempty"`  // SR-IOV support enables a hypervisor to create virtual instances of a PCI-express device, potentially increasing performance.  // This supports following options: true, false.  // +kubebuilder:validation:Enum=true;false  // 开启 sriov  SriovEnabled \*bool `json:"sriovEnabled,omitempty"`  } |

### HardwareProfile

|  |
| --- |
| HardwareProfile string `json:"hardwareProfile,omitempty"` |

### RootDeviceHints

有多块盘时，指定系统盘，参考：[Root device hints — Ironic Specs 0.0.1.dev739 documentation (openstack.org)](https://specs.openstack.org/openstack/ironic-specs/specs/kilo-implemented/root-device-hints.html#:~:text=When%20the%20deploy%20ramdisk%20boots%20Ironic%20picks%20the,is%20arbitrary%20%5B%201%20%5D%20%5B%202%20%5D.)

|  |
| --- |
| RootDeviceHints \*RootDeviceHints `json:"rootDeviceHints,omitempty"` |

### BootMode

裸金属启动的方式：

|  |
| --- |
| BootMode BootMode `json:"bootMode,omitempty"` |

BootMode 定义为：

|  |
| --- |
| （apis/metal3.io/v1alpha1/baremetalhost\_types.go）  // BootMode is the boot mode of the system  // +kubebuilder:validation:Enum=UEFI;UEFISecureBoot;legacy  type BootMode string  // Allowed boot mode from metal3  const (  UEFI BootMode = "UEFI"  UEFISecureBoot BootMode = "UEFISecureBoot"  Legacy BootMode = "legacy"  DefaultBootMode BootMode = UEFI  ) |

### BootMACAddress

指定 pxe 启动网卡的 mac 地址。

|  |
| --- |
| BootMACAddress string `json:"bootMACAddress,omitempty"` |

### Online

期望是 power on 还是 power off。

|  |
| --- |
| Online bool `json:"online"` |

这里是期望的状态，BareMetalHostStatus 中 PoweredOn 是实际状态。

### ConsumerRef

标识该对象被哪个上层对象消耗。

|  |
| --- |
| ConsumerRef \*corev1.ObjectReference `json:"consumerRef,omitempty"` |

例如：

|  |
| --- |
| consumerRef:  apiVersion: infrastructure.cluster.x-k8s.io/v1beta1  kind: Metal3Machine  name: test1-controlplane-lm46w  namespace: metal3 |

### Image

指定裸金属部署使用的镜像。

|  |
| --- |
| Image \*Image `json:"image,omitempty"` |

Image 定义为：

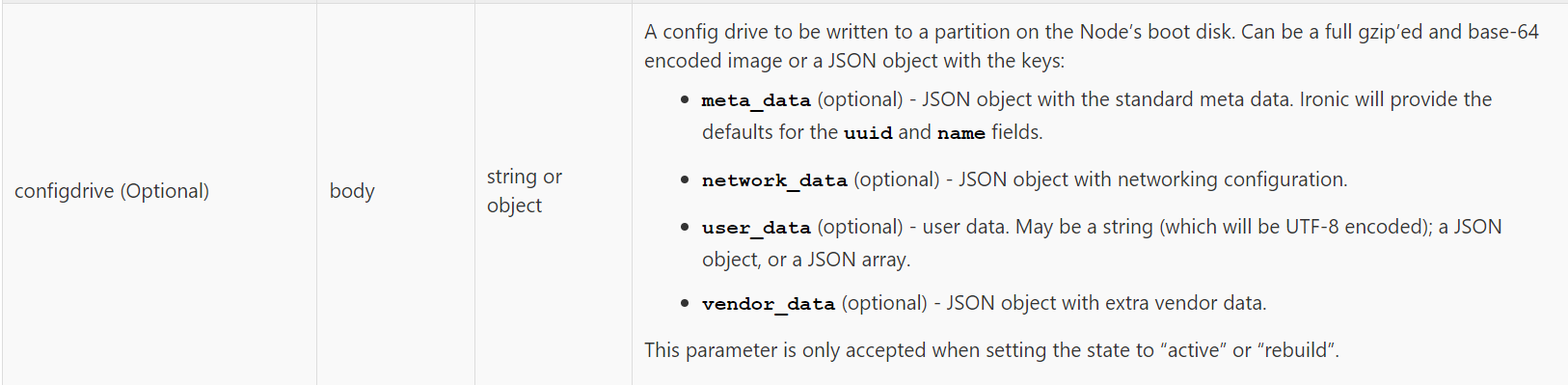
|  |
| --- |
| （apis/metal3.io/v1alpha1/baremetalhost\_types.go）  // Image holds the details of an image either to provisioned or that  // has been provisioned.  type Image struct {  // URL is a location of an image to deploy.  URL string `json:"url"`  // Checksum is the checksum for the image.  Checksum string `json:"checksum,omitempty"`  // ChecksumType is the checksum algorithm for the image.  // e.g md5, sha256, sha512  ChecksumType ChecksumType `json:"checksumType,omitempty"`  // DiskFormat contains the format of the image (raw, qcow2, ...).  // Needs to be set to raw for raw images streaming.  // Note live-iso means an iso referenced by the url will be live-booted  // and not deployed to disk, and in this case the checksum options  // are not required and if specified will be ignored.  // +kubebuilder:validation:Enum=raw;qcow2;vdi;vmdk;live-iso  DiskFormat \*string `json:"format,omitempty"`  } |

Image 示例：

|  |
| --- |
| image:  checksum: http://172.22.0.1/images/UBUNTU\_22.04\_NODE\_IMAGE\_K8S\_v1.27.1-raw.img.sha256sum  checksumType: sha256  format: raw  url: http://172.22.0.1/images/UBUNTU\_22.04\_NODE\_IMAGE\_K8S\_v1.27.1-raw.img |

### UserData&NetworkData&MetaData

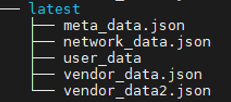
这三个成员都是 configdrive 相关的，其实对应的是 ironic 中的 api：/v1/nodes/{node\_ident}/states/provision （[Bare Metal API — Ironic API Reference documentation (openstack.org)](https://docs.openstack.org/api-ref/baremetal/?expanded=change-node-provision-state-detail)）中的 configdrive 字段参数：



在metal3 中，把 UserData、NetworkData 和 MetaData 拆分成了引用3个 Secret 对象：

|  |
| --- |
| UserData \*corev1.SecretReference `json:"userData,omitempty"  NetworkData \*corev1.SecretReference `json:"networkData,omitempty"`  MetaData \*corev1.SecretReference `json:"metaData,omitempty"` |

每个Secret 对象中包含 base64 编码的数据，会在裸金属部署时，写入到 config drive 中。这里参考 configdrive 的格式定义（参考 [Config drive - cloud-init 23.2 documentation (cloudinit.readthedocs.io)](https://cloudinit.readthedocs.io/en/latest/reference/datasources/configdrive.html)）：



MetaData 对应 meta\_data.json，NetworkData 对应 network\_data.json，UserData 对应 user\_data 文件。

#### Meta3 中的处理代码

这里首先看下 metal3 中的处理代码：

|  |
| --- |
| （pkg/provisioner/ironic/ironic.go）  func (p \*ironicProvisioner) getConfigDrive(data provisioner.ProvisionData) (configDrive nodes.ConfigDrive, err error) {  // In theory, Ironic can support configdrive with live ISO by attaching  // it to another virtual media slot. However, some hardware does not  // support two virtual media devices at the same time, so we shouldn't  // try it.  if data.Image.IsLiveISO() {  p.log.Info("not providing config drive for live ISO")  return  }  // Retrieve instance specific user data (cloud-init, ignition, etc).  userData, err := data.HostConfig.UserData()  if err != nil {  return configDrive, errors.Wrap(err, "could not retrieve user data")  }  if userData != "" {  configDrive.UserData = userData  }  // Retrieve OpenStack network\_data. Default value is empty.  networkDataRaw, err := data.HostConfig.NetworkData()  if err != nil {  return configDrive, errors.Wrap(err, "could not retrieve network data")  }  if networkDataRaw != "" {  var networkData map[string]interface{}  if err = yaml.Unmarshal([]byte(networkDataRaw), &networkData); err != nil {  return configDrive, errors.Wrap(err, "failed to unmarshal network\_data.json from secret")  }  configDrive.NetworkData = networkData  }  // Retrieve meta data with fallback to defaults from provisioner.  configDrive.MetaData = map[string]interface{}{  "uuid": string(p.objectMeta.UID),  "metal3-namespace": p.objectMeta.Namespace,  "metal3-name": p.objectMeta.Name,  "local-hostname": p.objectMeta.Name,  "local\_hostname": p.objectMeta.Name,  "name": p.objectMeta.Name,  }  metaDataRaw, err := data.HostConfig.MetaData()  if err != nil {  return configDrive, errors.Wrap(err, "could not retrieve metadata")  }  if metaDataRaw != "" {  if err = yaml.Unmarshal([]byte(metaDataRaw), &configDrive.MetaData); err != nil {  return configDrive, errors.Wrap(err, "failed to unmarshal metadata from secret")  }  }  return  } |

分别获取三部分数据，保存到 ConfigDrive 对象中。

在 ironicProvisioner.Provision() 中，传递给 ironic：

|  |
| --- |
| （pkg/provisioner/ironic/ironic.go）  func (p \*ironicProvisioner) Provision(data provisioner.ProvisionData, forceReboot bool) (result provisioner.Result, err error) {  …  case nodes.Available:  …  configDrive, err := p.getConfigDrive(data)  if err != nil {  return transientError(err)  }  return p.changeNodeProvisionState(  ironicNode,  nodes.ProvisionStateOpts{  Target: nodes.TargetActive,  ConfigDrive: configDrive,  DeploySteps: p.getCustomDeploySteps(data.CustomDeploy),  },  ) |

#### Ironic 中的处理代码

Api 接口的处理代码：

NodeStatesController.provision()

|  |
| --- |
| (ironic/api/controllers/v1/node.py)  def provision(self, node\_ident, target, configdrive=None,  clean\_steps=None, deploy\_steps=None,  rescue\_password=None, disable\_ramdisk=None):  …  m = ir\_states.machine.copy()  m.initialize(rpc\_node.provision\_state)  …  # 调用 \_do\_provision\_action()  self.\_do\_provision\_action(rpc\_node, target, configdrive, clean\_steps,  deploy\_steps, rescue\_password,  disable\_ramdisk) |

Configdrive 参数继续向下传。

NodeStatesController. \_do\_provision\_action()

|  |
| --- |
| (ironic/api/controllers/v1/node.py)  def \_do\_provision\_action(self, rpc\_node, target, configdrive=None,  clean\_steps=None, deploy\_steps=None,  rescue\_password=None, disable\_ramdisk=None):  …  if target in (ir\_states.ACTIVE, ir\_states.REBUILD, ir\_states.DEPLOY):  rebuild = (target == ir\_states.REBUILD)  if deploy\_steps:  \_check\_deploy\_steps(deploy\_steps)  api.request.rpcapi.do\_node\_deploy(context=api.request.context,  node\_id=rpc\_node.uuid,  rebuild=rebuild,  configdrive=configdrive,  topic=topic,  deploy\_steps=deploy\_steps) |

deployment.do\_node\_deploy()：

|  |
| --- |
| （ironic/conductor/deployments.py）  def do\_node\_deploy(task, conductor\_id=None, configdrive=None,  deploy\_steps=None):  """Prepare the environment and deploy a node."""  node = task.node  utils.wipe\_deploy\_internal\_info(task)  try:  if configdrive:  \_store\_configdrive(node, configdrive)  … |

其中，\_store\_configdrive() 先把 api 中指定的 configdrive 信息保存到数据库，如果指定了保存到 swift，会把 configdrive 信息保存到 swift，另外，也会保存到 ironic 数据库的 node.instance\_info[‘configdrive’] 字段中。

在 AgentDeploy 的 write\_image() 阶段：

|  |
| --- |
| （ironic/drivers/modules/agent.py）  @METRICS.timer('AgentDeployMixin.write\_image')  @base.deploy\_step(priority=80)  @task\_manager.require\_exclusive\_lock  def write\_image(self, task):  …  configdrive = manager\_utils.get\_configdrive\_image(node)  if configdrive:  # FIXME(dtantsur): remove this duplication once IPA is ready:  # https://review.opendev.org/c/openstack/ironic-python-agent/+/790471  image\_info['configdrive'] = configdrive  # Now switch into the corresponding in-band deploy step and let the  # result be polled normally.  // 把 configdrive 信息传给 ipa  new\_step = {'interface': 'deploy',  'step': 'write\_image',  'args': {'image\_info': image\_info,  'configdrive': configdrive}}  client = agent\_client.get\_client(task)  return agent\_base.execute\_step(task, new\_step, 'deploy',  client=client) |

get\_configdrive\_image()：

|  |
| --- |
| （ironic/conductor/utils.py）  def get\_configdrive\_image(node):  """  """  # 获取 \_store\_configdrive() 保存在 node 数据库中的 configdrive  configdrive = node.instance\_info.get('configdrive')  if isinstance(configdrive, dict):  configdrive = build\_configdrive(node, configdrive)  return configdrive |

build\_configdrive()：

|  |
| --- |
| （ironic/conductor/utils.py）  def build\_configdrive(node, configdrive):  """  """  meta\_data = configdrive.setdefault('meta\_data', {})  meta\_data.setdefault('uuid', node.uuid)  if node.name:  meta\_data.setdefault('name', node.name)  user\_data = configdrive.get('user\_data')  if isinstance(user\_data, (dict, list)):  user\_data = jsonutils.dump\_as\_bytes(user\_data)  elif user\_data:  user\_data = user\_data.encode('utf-8')  LOG.debug('Building a configdrive for node %s', node.uuid)  # 这个模块函数在 openstacksdk 的包中  return os\_configdrive.build(meta\_data, user\_data=user\_data,  network\_data=configdrive.get('network\_data'),  vendor\_data=configdrive.get('vendor\_data')) |

在 openstacksdk 的这个函数中，会把这些 data 信息写入一个 iso image 中，并且打上 “config-2” 的标签。

### PreprovisioningNetworkDataName

传递给 preprovision image 的网络配置（network\_data.json）

|  |
| --- |
| PreprovisioningNetworkDataName string `json:"preprovisioningNetworkDataName,omitempty"` |

### Description

人为增加的描述信息，没有其他作用。

|  |
| --- |
| Description string `json:"description,omitempty"` |

### ExternallyProvisioned

设置为true时，由外部进行裸金属的部署，参考 ironic 的 adopt。

|  |
| --- |
| ExternallyProvisioned bool `json:"externallyProvisioned,omitempty"` |

### AutomatedCleaningMode

|  |
| --- |
| AutomatedCleaningMode AutomatedCleaningMode `json:"automatedCleaningMode,omitempty"` |

### CustomDeploy

|  |
| --- |
| CustomDeploy \*CustomDeploy `json:"customDeploy,omitempty"` |

## BareMetalHostStatus

BareMetalHostStatus 是程序维护的部分。

### OperationalStatus

### ErrorType

### LastUpdated

### HardwareProfile

### HardwareDetails

表示裸金属机器的硬件信息。

|  |
| --- |
| HardwareDetails \*HardwareDetails `json:"hardware,omitempty"` |

HardwareDetails 定义为：

|  |
| --- |
| （apis/metal3.io/v1alpha1/baremetalhost\_types.go）  // HardwareDetails collects all of the information about hardware  // discovered on the host.  type HardwareDetails struct {  SystemVendor HardwareSystemVendor `json:"systemVendor,omitempty"`  Firmware Firmware `json:"firmware,omitempty"`  RAMMebibytes int `json:"ramMebibytes,omitempty"`  // 每个 NIC 表示一块网卡信息  NIC []NIC `json:"nics,omitempty"`  // 每个 Storage 对象表示一块盘的信息  Storage []Storage `json:"storage,omitempty"`  CPU CPU `json:"cpu,omitempty"`  Hostname string `json:"hostname,omitempty"`  } |

#### CPU

表示 cpu 信息：

|  |
| --- |
| （apis/metal3.io/v1alpha1/baremetalhost\_types.go）  // CPU describes one processor on the host.  type CPU struct {  // cpu 的架构  Arch string `json:"arch,omitempty"`  // cpu 的型号  Model string `json:"model,omitempty"`  ClockMegahertz ClockSpeed `json:"clockMegahertz,omitempty"`  // cpu 支持的功能  Flags []string `json:"flags,omitempty"`  // cpu 个数  Count int `json:"count,omitempty"`  } |

下面是 cpu 的一个示例：

|  |
| --- |
| cpu:  arch: x86\_64  count: 2  flags:  - 3dnowprefetch  - abm  - adx  - aes  - apic  - arat  - arch\_capabilities  - arch\_perfmon  - …  model: Intel(R) Xeon(R) Gold 5318Y CPU @ 2.10GHz |

#### RAMMebibytes

内存，这里只采集了一个内存大小。

#### Storage

表示磁盘信息。Storage 定义为：

|  |
| --- |
| （apis/metal3.io/v1alpha1/baremetalhost\_types.go）  // Storage describes one storage device (disk, SSD, etc.) on the host.  type Storage struct {  // The Linux device name of the disk, e.g. "/dev/sda". Note that this  // may not be stable across reboots.  Name string `json:"name,omitempty"`  // Whether this disk represents rotational storage.  // This field is not recommended for usage, please  // prefer using 'Type' field instead, this field  // will be deprecated eventually.  Rotational bool `json:"rotational,omitempty"`  // Device type, one of: HDD, SSD, NVME.  // +kubebuilder:validation:Optional  // +kubebuilder:validation:Enum=HDD;SSD;NVME;  Type DiskType `json:"type,omitempty"`  // The size of the disk in Bytes  SizeBytes Capacity `json:"sizeBytes,omitempty"`  // The name of the vendor of the device  Vendor string `json:"vendor,omitempty"`  // Hardware model  Model string `json:"model,omitempty"`  // The serial number of the device  SerialNumber string `json:"serialNumber,omitempty"`  // The WWN of the device  WWN string `json:"wwn,omitempty"`  // The WWN Vendor extension of the device  WWNVendorExtension string `json:"wwnVendorExtension,omitempty"`  // The WWN with the extension  WWNWithExtension string `json:"wwnWithExtension,omitempty"`  // The SCSI location of the device  HCTL string `json:"hctl,omitempty"`  } |

Storage 示例为：

|  |
| --- |
| storage:  - hctl: "0:0:0:0"  model: QEMU HARDDISK  name: /dev/disk/by-path/pci-0000:03:00.0-scsi-0:0:0:0  rotational: true  serialNumber: drive-scsi0-0-0-0  sizeBytes: 53687091200  type: HDD  vendor: QEMU |

#### NIC

NIC 表示一块网卡信息，定义为：

|  |
| --- |
| （apis/metal3.io/v1alpha1/baremetalhost\_types.go）  // NIC describes one network interface on the host.  type NIC struct {  // The name of the network interface, e.g. "en0"  Name string `json:"name,omitempty"`  // The vendor and product IDs of the NIC, e.g. "0x8086 0x1572"  Model string `json:"model,omitempty"`  // The device MAC address  // +kubebuilder:validation:Pattern=`[0-9a-fA-F]{2}(:[0-9a-fA-F]{2}){5}`  MAC string `json:"mac,omitempty"`  // The IP address of the interface. This will be an IPv4 or IPv6 address  // if one is present. If both IPv4 and IPv6 addresses are present in a  // dual-stack environment, two nics will be output, one with each IP.  IP string `json:"ip,omitempty"`  // The speed of the device in Gigabits per second  SpeedGbps int `json:"speedGbps,omitempty"`  // The VLANs available  VLANs []VLAN `json:"vlans,omitempty"`  // The untagged VLAN ID  VLANID VLANID `json:"vlanId,omitempty"`  // Whether the NIC is PXE Bootable  PXE bool `json:"pxe,omitempty"`  } |

NIC 的示例：

|  |
| --- |
| nics:  - ip: 172.22.0.75  mac: 00:2a:ec:ab:5f:15  model: 0x1af4 0x0001  name: enp1s0  pxe: true  - ip: fe80::1840:51a0:94af:73b1%enp1s0  mac: 00:2a:ec:ab:5f:15  model: 0x1af4 0x0001  name: enp1s0  pxe: true  - ip: 192.168.111.21  mac: 00:2a:ec:ab:5f:17  model: 0x1af4 0x0001  name: enp2s0  - ip: fe80::e1d9:cead:2169:b1e8%enp2s0  mac: 00:2a:ec:ab:5f:17  model: 0x1af4 0x0001  name: enp2s0 |

#### HardwareDetails 是如何采集的？

关于 HardwareDetails 的一个重要问题是是如何采集的。InspectHardware 在 Provisioner 的 InspectHardware() 接口中获取：

|  |
| --- |
| （pkg/provisioner/provisioner.go）  type Provisioner interface {  …  // InspectHardware updates the HardwareDetails field of the host with  // details of devices discovered on the hardware. It may be called  // multiple times, and should return true for its dirty flag until the  // inspection is completed.  // HardwareDetails 作为 InspectHardware 接口的返回值  InspectHardware(data InspectData, restartOnFailure, refresh, forceReboot bool) (result Result, started bool, details \*metal3v1alpha1.HardwareDetails, err error)  …  } |

对于 ironic 来说，它是进入 ironic 的 Inspect 阶段，把 ironic 的结果转成 HardwareDetails。

首先是 ironic 的 provisioner 的 InspectHardware() 中，从 ironic 获取inpect 的信息，然后通过 GetHardwareDetails() 转成 HardwareDetails 返回：

|  |
| --- |
| （pkg/provisioner/ironic/ironic.go）  func (p \*ironicProvisioner) InspectHardware(data provisioner.InspectData, restartOnFailure, refresh, forceReboot bool) (result provisioner.Result, started bool, details \*metal3v1alpha1.HardwareDetails, err error) {  …  // 从 ironic 获取 inspect 结果  introData, err := response.Extract()  …  // 转成 HardwareDetails 返回  details = hardwaredetails.GetHardwareDetails(introData)  p.publisher("InspectionComplete", "Hardware inspection completed")  result, err = operationComplete()  return  } |

在 BareMetalHostReconciler.actionInspecting() 中生成一个 HardwareData 对象，保存到 k8s：

|  |
| --- |
| （controllers/metal3.io/baremetalhost\_controller.go）  // Ensure we have the information about the hardware on the host.  func (r \*BareMetalHostReconciler) actionInspecting(prov provisioner.Provisioner, info \*reconcileInfo) actionResult {  …  // Create HardwareData with the same name and namesapce as BareMetalHost  hardwareData := &metal3v1alpha1.HardwareData{}  hardwareDataKey := client.ObjectKey{  Name: info.host.Name,  Namespace: info.host.Namespace,  }  hd := &metal3v1alpha1.HardwareData{  TypeMeta: metav1.TypeMeta{  Kind: "HardwareData",  APIVersion: metal3v1alpha1.GroupVersion.String(),  },  ObjectMeta: metav1.ObjectMeta{  Name: info.host.Name,  Namespace: info.host.Namespace,  // Register the finalizer immediately  Finalizers: []string{  hardwareDataFinalizer,  },  OwnerReferences: []metav1.OwnerReference{  \*metav1.NewControllerRef(info.host, metal3v1alpha1.GroupVersion.WithKind("BareMetalHost")),  },  },  Spec: metal3v1alpha1.HardwareDataSpec{  HardwareDetails: details,  },  }  // either hardwareData was deleted above, or not found. We need to re-create it  if err := r.Client.Create(context.Background(), hd); err != nil {  return actionError{errors.Wrap(err, "failed to create hardwareData")}  }  } |

在 BareMetalHostReconciler.reconciletHostData() 中，把 HardwareData 中的HardwareDetails更新到 BareMetalHost.Status.HardwareDetails 中。

### ProvisionStatus

这个字段比较重要，保存了裸金属的状态信息。

|  |
| --- |
| // Information tracked by the provisioner.  Provisioning ProvisionStatus `json:"provisioning"` |

ProvisionStatus 定义为：

|  |
| --- |
| （apis/metal3.io/v1alpha1/baremetalhost\_types.go）  // ProvisionStatus holds the state information for a single target.  type ProvisionStatus struct {  // An indiciator for what the provisioner is doing with the host.  // 裸金属的状态  State ProvisioningState `json:"state"`  // The machine's UUID from the underlying provisioning tool  // 裸金属在 provisioner 中的唯一 uuid，例如 ironic 中的 node id  ID string `json:"ID"`  // Image holds the details of the last image successfully  // provisioned to the host.  // 镜像的信息  Image Image `json:"image,omitempty"`  // The RootDevicehints set by the user  RootDeviceHints \*RootDeviceHints `json:"rootDeviceHints,omitempty"`  // BootMode indicates the boot mode used to provision the node  // 当前裸金属的 bootMode  BootMode BootMode `json:"bootMode,omitempty"`  // The Raid set by the user  RAID \*RAIDConfig `json:"raid,omitempty"`  // The Bios set by the user  Firmware \*FirmwareConfig `json:"firmware,omitempty"`  // Custom deploy procedure applied to the host.  CustomDeploy \*CustomDeploy `json:"customDeploy,omitempty"`  } |

#### ProvisionStatus 的更新逻辑

##### ProvisioningState 的更新

ProvisioningState 的更新在 hostStateMachine 这个 fsm 状态机中。在 ReconcileState() 中：

|  |
| --- |
| （controllers/metal3.io/host\_state\_machine.go）  func (hsm \*hostStateMachine) ReconcileState(info \*reconcileInfo) (actionRes actionResult) {  initialState := hsm.Host.Status.Provisioning.State  defer func() {  if overrideAction := hsm.updateHostStateFrom(initialState, info); overrideAction != nil {  actionRes = overrideAction  }  }()  if delayedResult := hsm.checkDelayedHost(info); delayedResult != nil {  return delayedResult  }  if hsm.checkInitiateDelete(info.log) {  info.log.Info("Initiating host deletion")  return actionComplete{}  }  if detachedResult := hsm.checkDetachedHost(info); detachedResult != nil {  return detachedResult  }  if registerResult := hsm.ensureRegistered(info); registerResult != nil {  hostRegistrationRequired.Inc()  return registerResult  }  // 调用当前状态的 handler() 处理，得到目标状态 NextState  if stateHandler, found := hsm.handlers()[initialState]; found {  return stateHandler(info)  }  info.log.Info("No handler found for state", "state", initialState)  return actionError{fmt.Errorf("No handler found for state \"%s\"", initialState)}  } |

这里状态流转的重点在于 updateHostStateFrom() 这个 defer 函数：

|  |
| --- |
| （controllers/metal3.io/host\_state\_machine.go）  func (hsm \*hostStateMachine) updateHostStateFrom(initialState metal3v1alpha1.ProvisioningState,  info \*reconcileInfo) actionResult {  // 如果 handler 中处理完成，会设置 NextState  if hsm.NextState != initialState {  …  hsm.Host.Status.Provisioning.State = hsm.NextState  …  }  } |

如果 initialState 是当前状态，NextState 是 handler 设置的下一个状态。

### GoodCredentials

### TriedCredentials

### ErrorMessage

### PoweredOn

裸金属实际的电源状态。

### OperationHistory

### ErrorCount

# 问题解决

## （1）metal3 的 ramdisk 小系统的 http 地址是怎么配置的？

# Metal3Machine

## Metal3Machine 和 BareMetalHost 的关联

Metal3Machine 上包含一个 HostAnnotation：

"metal3.io/BareMetalHost" 指定其关联的 BareMetalHost。示例：

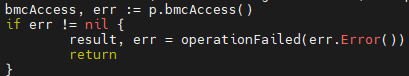
|  |
| --- |
| - apiVersion: infrastructure.cluster.x-k8s.io/v1beta1  kind: Metal3Machine  metadata:  annotations:  cluster.x-k8s.io/cloned-from-groupkind: Metal3MachineTemplate.infrastructure.cluster.x-k8s.io  cluster.x-k8s.io/cloned-from-name: test1-controlplane  metal3.io/BareMetalHost: metal3/node-1 |

BareMetalHost 的调度，在 MachineManager.Associate() 中。

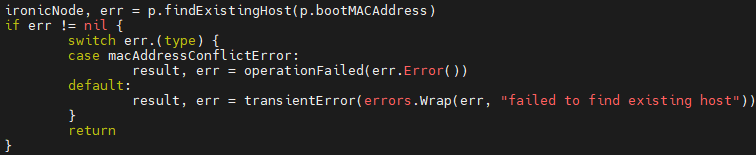
# 函数跟踪

## ironicProvisioner.ValidateManagementAccess()

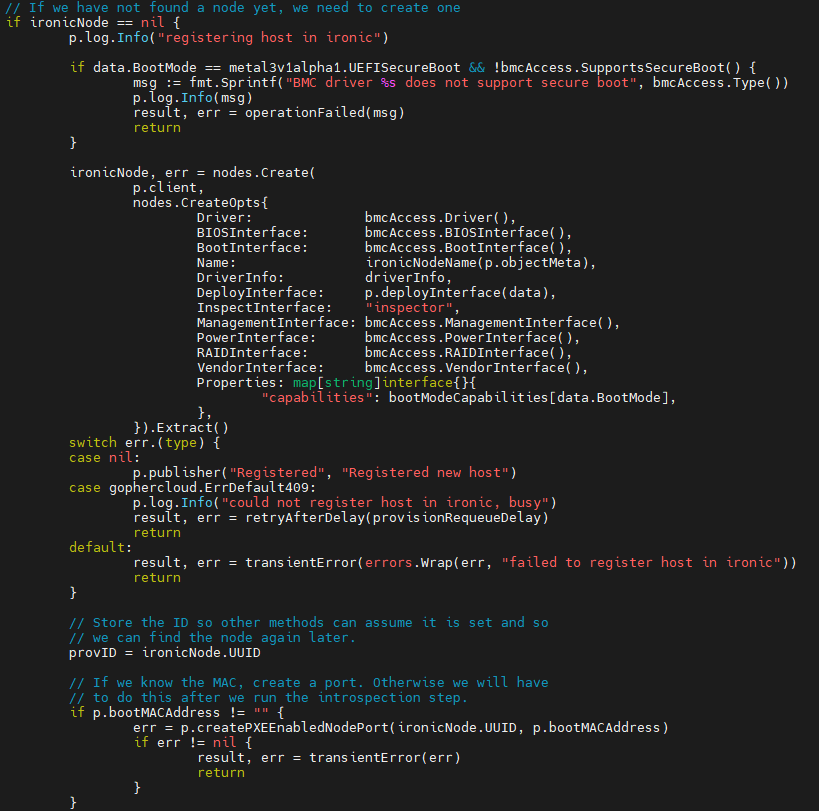
|  |
| --- |
| （pkg/provisioner/ironic/ironic.go）  // 参数中，data 是 metal3 传给 provisioner 的数据  // 返回值：  // result：返回操作结果  // provID：裸金属在 provisioner 中的唯一 uuid，参考 BareMetalHostStatus.ProvisionStatus.ID 字段  func (p \*ironicProvisioner) ValidateManagementAccess(data provisioner.ManagementAccessData, credentialsChanged, restartOnFailure bool) (result provisioner.Result, provID string, err error) { |



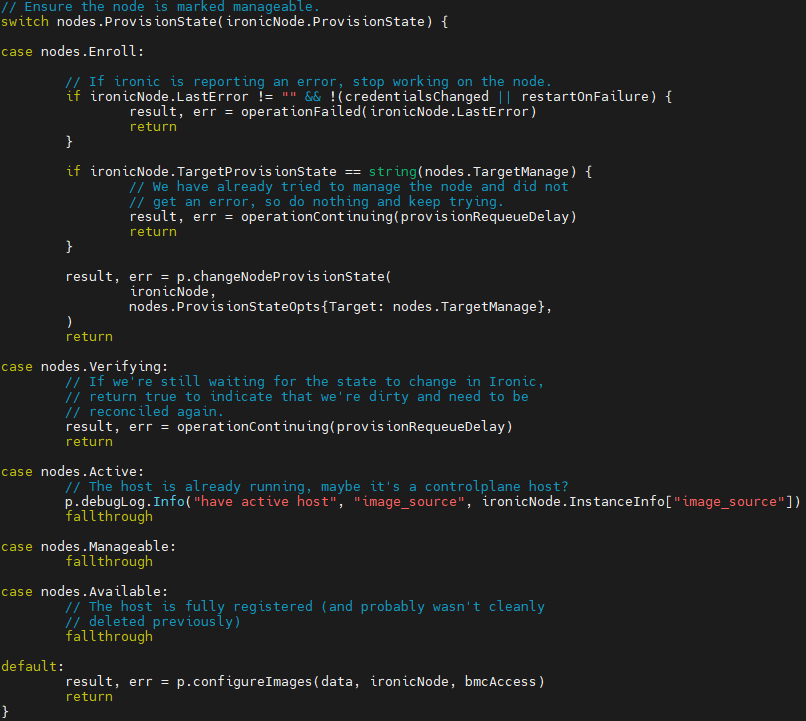
初始化 BMC 访问信息。



根据 mac 地址查找已经存在的 node。后面会根据查找的结果，如果不存在，则创建，存在则更新。



在ironic 上创建一个 node。



将 ironicNode 设置为 manageable 状态。并且，在 manageable 状态下，调用 configureImages() 配置 Image。

