弹性训练发现节点 & State

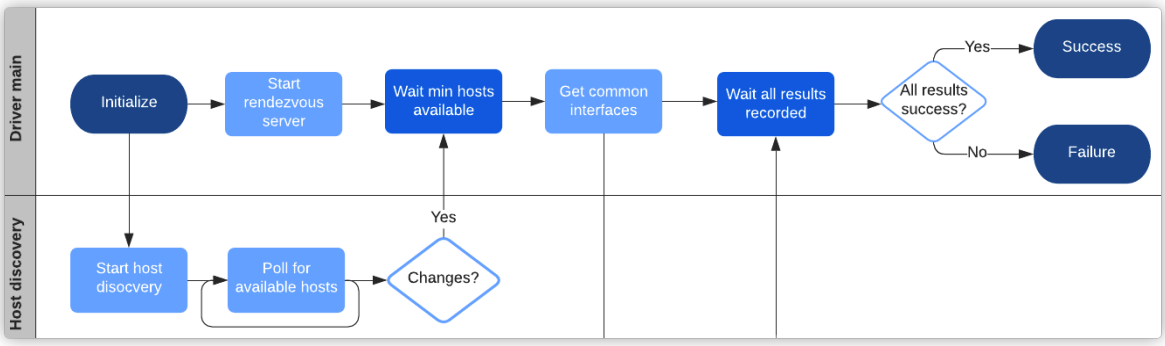
目录

0x00 摘要

看看horovod 如何动态发现节点 和 状态信息。

0x01 设计点

本文对应架构图中的 Host Discovery 部分，因为是被 Driver Main 调用，所以把两部分一起展示出。



发现节点机制的几个关键设计点如下：

- **有节点变化时候，如何即时发现**？Horovod是通过定期调用完成。
- **发现节点变化时候，如何通知各个worker**？Horovod通过构建了一个通知机制完成。即，每个worker把自己注册到WorkerNotificationManager 之上，当有节点变化时候，WorkerNotificationManager 会逐一通知这些worker。
- **worker得到通知之后，如何处理**？Horovod 把worker的状态在深度框架上进一步封装成各种State，得到通知之后就会调用State的对应callback函数，或者同步状态，或者进行其他处理。

0x02 发现机制

这部分代码主要在：horovod/runner/elastic/discovery.py。

2.1 发现脚本

HostDiscoveryScript 的主要作用就是保存脚本（程序启动时候设置进来），然后当执行 find_available_hosts_and_slots 的时候，调用这个发现脚本，得到 host 信息。

该脚本的输出的格式 就是调用 horovodrun 时候 的 host 参数格式，比如：

```
$ sh ./discover_hosts.sh    # 运行脚本，输出节点信息
10.68.32.2:4
10.68.32.3:4
10.68.32.4:4
```

定义如下：

```
class HostDiscoveryScript(HostDiscovery):

    def __init__(self, discovery_script, slots):
        self._discovery_script = discovery_script # 设定脚本
        self._default_slots = slots # 审定slots
        super(HostDiscoveryScript, self).__init__()

    def find_available_hosts_and_slots(self):
        stdout = io.StringIO()
        # 执行发现脚本
        exit_code = safe_shell_exec.execute(self._discovery_script,
stdout=stdout)

        # 读取脚本输出，解析出来host信息
        host_slots = {}
        lines = set(stdout.getvalue().strip().split('\n'))
        for line in lines:
            host = line
            if ':' in line:
                host, slots = line.split(':')
                host_slots[host] = int(slots)
            else:
                host_slots[host] = self._default_slots
        return host_slots
```

2.2 HostManager

HostManager 是 host discovery 的核心，作用是维护当前 host 以及 状态，其主要变量是：

- self._current_hosts： 当前的 host 信息，包括 slot，assign order 等等；
- self._hosts_state：当前的 host 状态，包括黑名单，event 等；
- self._discovery：可以认为是对 发现脚本 的一个封装，用来动态执行 发现脚本，获取 host 信息；

```
class HostManager(object):
    def __init__(self, discovery):
        self._current_hosts = DiscoveredHosts(host_slots={},
host_assignment_order=[])
        self._hosts_state = defaultdict(HostState)
        self._discovery = discovery

    def update_available_hosts(self):
        # TODO(travis): also check for hosts removed from the blacklist in the
future
        # 检查更新，给出是添加，还是删除节点
        def check_update(cur_host_slots, prev_host_slots):
            res = HostUpdateResult.no_update

            for prev_h in prev_host_slots:
                if prev_h not in cur_host_slots:
                    # prev_h is a removed host
                    res |= HostUpdateResult.removed

            for h in cur_host_slots:
                if h not in prev_host_slots:
                    # h is an added host
                    res |= HostUpdateResult.added
                elif cur_host_slots[h] > prev_host_slots[h]:
                    # h has more slots added
                    res |= HostUpdateResult.added
                elif cur_host_slots[h] < prev_host_slots[h]:
                    # h has removed some slots
                    res |=  HostUpdateResult.removed
            return res

        prev_host_slots = self._current_hosts.host_slots
        prev_host_assignment_order = self._current_hosts.host_assignment_order
        host_slots = self._discovery.find_available_hosts_and_slots()

        if prev_host_slots != host_slots: # 有修改
            # 找到不在黑名单里的host
            available_hosts = set([host for host in host_slots.keys() if not
self._hosts_state[host].is_blacklisted()])
            # 找到host的order
            host_assignment_order =
HostManager.order_available_hosts(available_hosts, prev_host_assignment_order)
            self._current_hosts = DiscoveredHosts(host_slots=host_slots,

 host_assignment_order=host_assignment_order)
            # 检查更新
            return check_update(self._current_hosts.host_slots, prev_host_slots)
        else: # 没修改就不更新
```

```
            return HostUpdateResult.no_update
```

HostManager 核心逻辑是 update_available_hosts 方法，就是用来发现可用的 host。

2.2.1 order_available_hosts

order_available_hosts 的作用是：确保最老的host被赋予最低的rank，即rank 0，因为最老的host最有可能拥有原来训练的模型以及训练状态，这些信息需要在下一轮新迭代之前，发给所有worker。

```
    @staticmethod
    def order_available_hosts(available_hosts, prev_host_assignment_order):
        # We need to ensure this list preserves relative order to ensure the
oldest hosts are assigned lower ranks.
        host_assignment_order = [host for host in prev_host_assignment_order if
host in available_hosts]
        known_hosts = set(host_assignment_order)
        for host in available_hosts:
            if host not in known_hosts:
                host_assignment_order.append(host)
        return host_assignment_order
```

2.3 配置

我们看看是发现脚本如何配置进入HostManager之中。

首先，发现脚本是在_run_elastic之中配置。

```
def _run_elastic(args):
    # construct host discovery component
    if args.host_discovery_script:
        # 如果参数中有设置发现脚本，则赋值为discover_hosts
        discover_hosts =
discovery.HostDiscoveryScript(args.host_discovery_script, args.slots)
    elif args.hosts: # 如果参数设置好了hosts，则赋值为discover_hosts
        _, available_host_slots = hosts.parse_hosts_and_slots(args.hosts)
        if len(available_host_slots) < 2:
            raise ValueError('Cannot run in fault tolerance mode with fewer than
2 hosts.')
        discover_hosts = discovery.FixedHosts(available_host_slots)
    else: # 抛出异常
        raise ValueError('One of --host-discovery-script, --hosts, or --
hostnames must be provided')

    # 配置进入setting
    settings = elastic_settings.ElasticSettings(discovery=discover_hosts,
                                                .....)

    env = os.environ.copy()
    config_parser.set_env_from_args(env, args)
    gloo_run_elastic(settings, env, args.command)
```

其次，发现脚本被设置到ElasticSettings之中。

```
class ElasticSettings(BaseSettings):
    def __init__(self, discovery, min_np, max_np, elastic_timeout, reset_limit,
**kwargs):
        self.discovery = discovery
```

当启动时候，会设置到 ElasticDriver 之中。

```
def start(self):
    """Starts the Horovod driver and services."""
    self.rendezvous = RendezvousServer(self.settings.verbose)
    self.driver = ElasticDriver(
        rendezvous=self.rendezvous,
        discovery=self.settings.discovery, # 在这里设置发现脚本
        min_np=self.settings.min_np,
        max_np=self.settings.max_np,
        timeout=self.settings.elastic_timeout,
        reset_limit=self.settings.reset_limit,
        verbose=self.settings.verbose)
```

最后，建立HostManager时候，会设置发现脚本。

```
class ElasticDriver(object):
    def __init__(self, rendezvous, discovery, min_np, max_np, timeout=None,
reset_limit=None, verbose=0):
        self._rendezvous = rendezvous
        self._host_manager = HostManager(discovery) # 设置脚本
```

0x03 如何调用

3.1 无限循环线程

HostManager 的调用逻辑是在 ElasticDriver 类中。

ElasticDriver 在初始化时候，生成一个后台线程 _discovery_thread。

```
self._discovery_thread = threading.Thread(target=self._discover_hosts)
```

3.1.1 定时探寻

在 `_discovery_thread` 之中，会运行_discover_hosts。

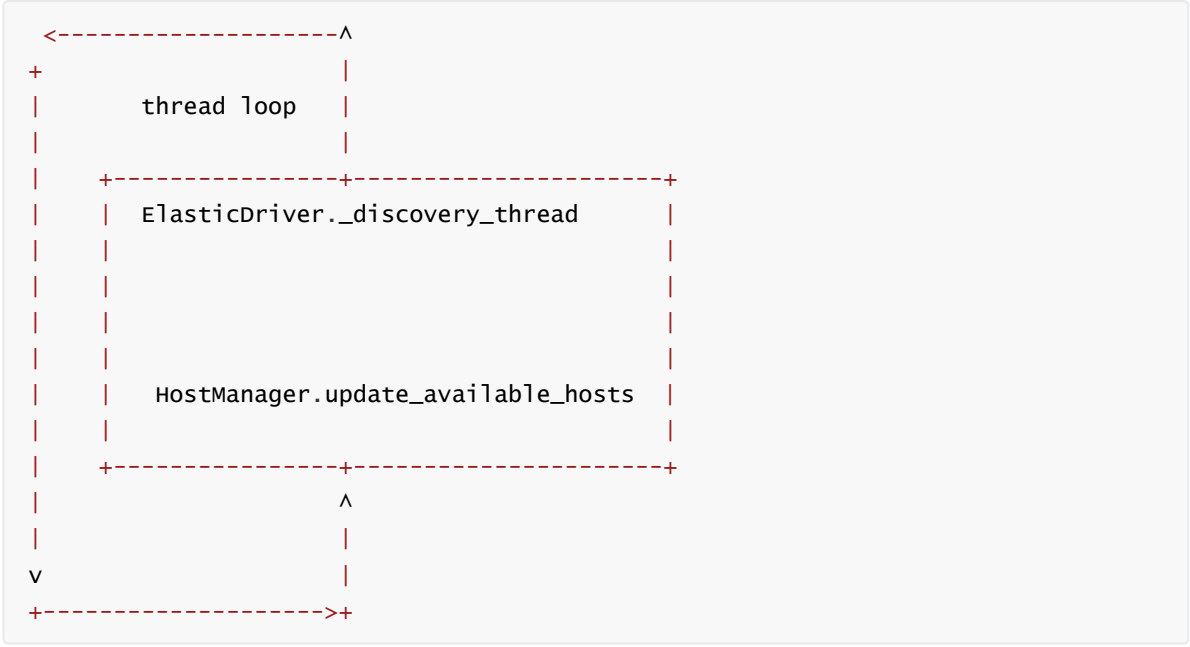
`ElasticDriver._discover_hosts` 会：

- 首先调用 `self._host_manager.update_available_hosts(self._host_manager.current_hosts, update_res)` 得到最新的host状态；
- 其次，如果新 host 状态已经发生的变化，于是就调用 _notify_workers_host_changes 和 _wait_hosts_cond.notify_all 来通知大家有 host 变化了；

```
def _discover_hosts(self):
    first_update = True
    while not self._shutdown.is_set():
        self._wait_hosts_cond.acquire()
        try:
```

```
            # 得到最新的host状态
            update_res = self._host_manager.update_available_hosts()
            if update_res != HostUpdateResult.no_update:

self._notify_workers_host_changes(self._host_manager.current_hosts, update_res)
                self._wait_hosts_cond.notify_all() # 通知大家有 host 变化
        except RuntimeError as e:
            if first_update:
                # Misconfiguration, fail the job immediately
                self._shutdown.set()
                self._wait_hosts_cond.notify_all() # 通知大家有 host 变化
                raise
            # Transient error, retry until timeout
            logging.warning(str(e))
        finally:
            self._wait_hosts_cond.release()
        first_update = False
        self._shutdown.wait(DISCOVER_HOSTS_FREQUENCY_SECS)
```

逻辑如下，是一个 thread loop 定时运行：

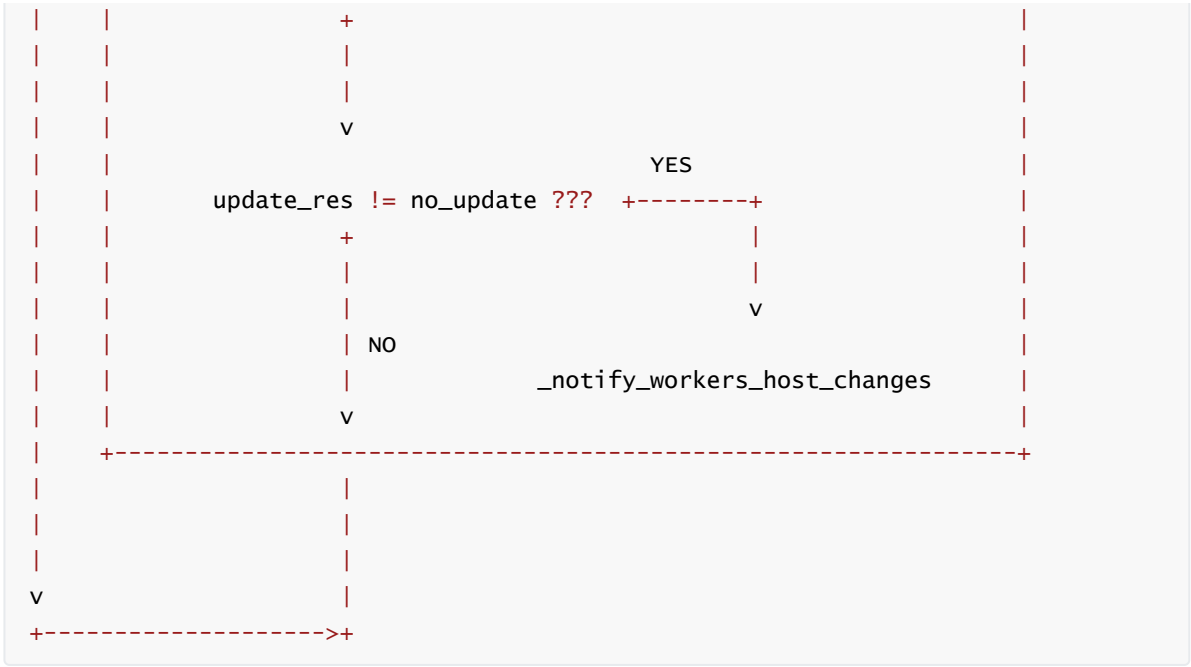


3.1.2 通知变化

如果发现有host 变化，则调用 `self._notify_workers_host_changes` 来通知。

即，**当Driver的定时进程通过节点发现脚本发现某一个节点被标记为新增或者移除时，它将 调用 _notify_workers_host_changes 发送一个通知到所有workers。**

逻辑如下：



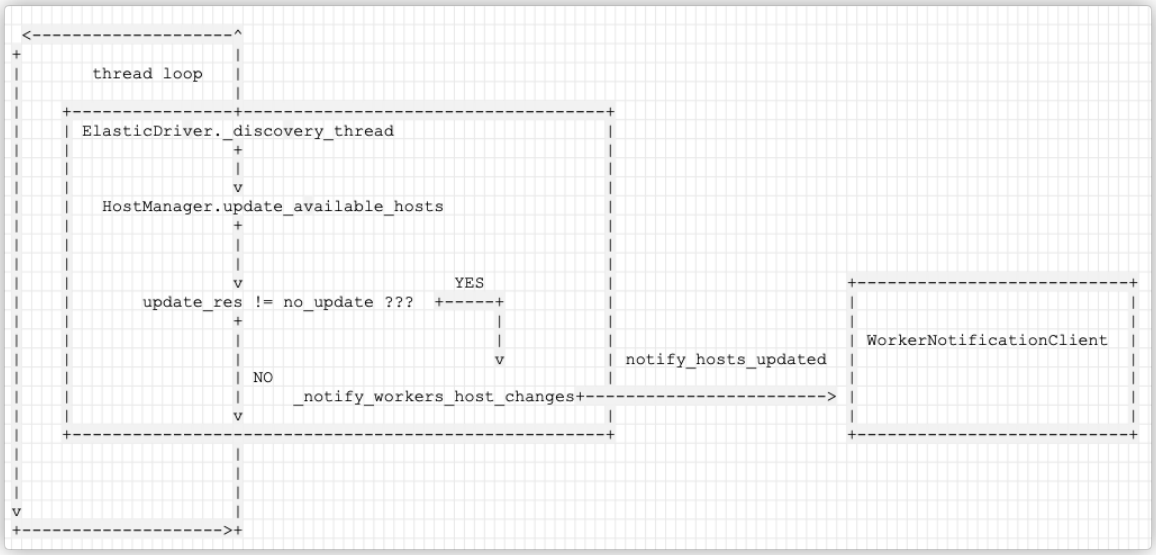
具体如下：

```
def _notify_workers_host_changes(self, current_hosts, update_res):
    next_host_assignments = {}
    if current_hosts.count_available_slots() >= self._min_np:
        # Assignments are required to be stable via contract
        next_host_assignments, _ = self._get_host_assignments(current_hosts)

    if next_host_assignments == self.host_assignments:
        # Skip notifying workers when host changes would not result in changes
of host assignments
        return

    coordinator_slot_info = self.get_coordinator_info()
    # 获取 WorkerNotificationClient
    coordinator_client = self.get_worker_client(coordinator_slot_info)

    timestamp = _epoch_time_s()
    coordinator_client.notify_hosts_updated(timestamp, update_res) # 通知
```

get_worker_client 函数就是获取 WorkerNotificationClient，然后调用 WorkerNotificationClient 来进行通知，所以下面我们接下来看 WorkerNotificationClient。

```
def get_worker_client(self, slot_info):
    return self._worker_clients.get((slot_info.hostname, slot_info.local_rank))
```

具体如下：



3.2 如何通知

就是利用 WorkerNotificationClient 发送 HostsUpdatedRequest。

3.2.1 WorkerNotificationClient

可以看到，WorkerNotificationService 继承了 network.BasicService，所以 WorkerNotificationClient 就是作为 WorkerNotificationService 的操作接口，从而给 WorkerNotificationService 发送 HostsUpdatedRequest。

```
class WorkerNotificationClient(network.BasicClient):
    def __init__(self, addresses, key, verbose, match_intf=False):
        super(WorkerNotificationClient,
self).__init__(WorkerNotificationService.NAME,
                                                    addresses,
                                                    key,
                                                    verbose,
                                                    match_intf=match_intf)

    def notify_hosts_updated(self, timestamp, update_res):
        self._send(HostsUpdatedRequest(timestamp, update_res))
```

3.2.2 WorkerNotificationService

WorkerNotificationService 会响应 HostsUpdatedRequest。

```
class WorkerNotificationService(network.BasicService):
    NAME = 'worker notification service'

    def __init__(self, key, nic, manager):
        super(WorkerNotificationService,
self).__init__(WorkerNotificationService.NAME,
                                                     key,
                                                     nic)
        self._manager = manager

    def _handle(self, req, client_address):
        if isinstance(req, HostsUpdatedRequest):
            self._manager.handle_hosts_updated(req.timestamp, req.res)
```

```
            return network.AckResponse()

        return super(WorkerNotificationService, self)._handle(req,
client_address)
```

3.2.3 WorkerNotificationManager

handle_hosts_updated 会逐一通知注册在WorkerNotificationManager 上的 listener（就是用户代码中的 State）。

WorkerNotificationManager 是在 horovod/common/elastic.py 构建，每一个host上运行一个。

```
notification_manager = WorkerNotificationManager()
```

具体定义如下：

```
class WorkerNotificationManager(object):
    def __init__(self):
        self._lock = threading.Lock()
        self._service = None
        self._listeners = set()

    def init(self, rendezvous_addr=None, rendezvous_port=None,
             nic=None, hostname=None, local_rank=None):
        with self._lock:
            if self._service:
                return

            rendezvous_addr = rendezvous_addr or
os.environ.get(HOROVOD_GLOO_RENDEZVOUS_ADDR)
            if not rendezvous_addr:
                return

            rendezvous_port = rendezvous_port if rendezvous_port is not None
else \
                int(os.environ.get(HOROVOD_GLOO_RENDEZVOUS_PORT))
            nic = nic or os.environ.get(HOROVOD_GLOO_IFACE)
            hostname = hostname or os.environ.get(HOROVOD_HOSTNAME)
            local_rank = local_rank if local_rank is not None else \
                int(os.environ.get(HOROVOD_LOCAL_RANK))

            secret_key = secret.make_secret_key()
            self._service = WorkerNotificationService(secret_key, nic, self)

            value = (self._service.addresses(), secret_key)
            put_data_into_kvstore(rendezvous_addr,
                                  rendezvous_port,
                                  PUT_WORKER_ADDRESSES,
                                  self._create_id(hostname, local_rank),
                                  value)

    def register_listener(self, listener):
        self._listeners.add(listener)

    def remove_listener(self, listener):
        self._listeners.remove(listener)
```

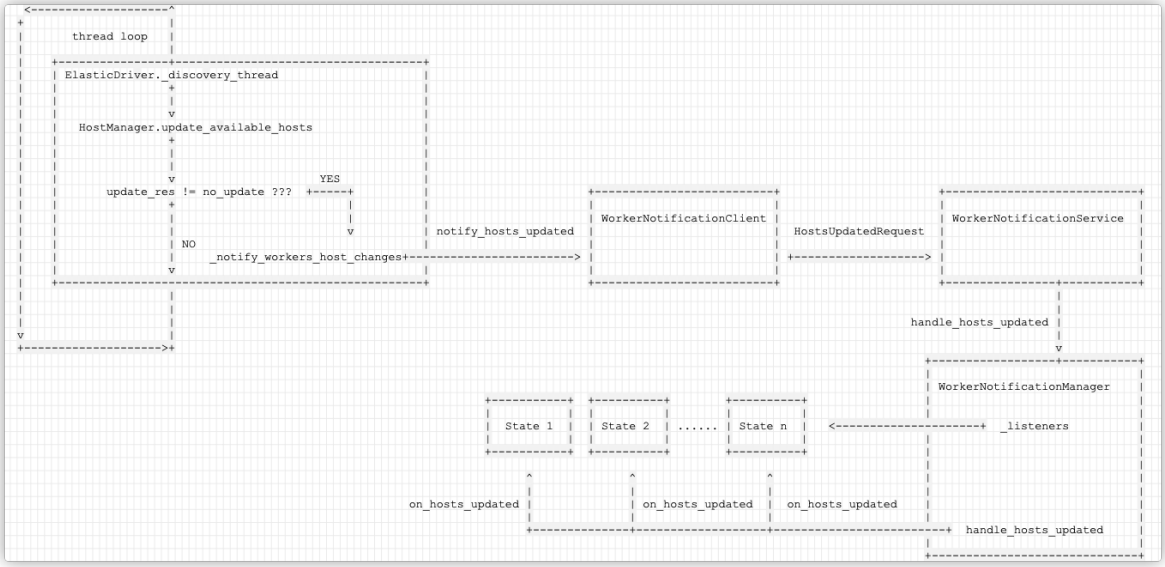
```
    def handle_hosts_updated(self, timestamp, update_res):
        for listener in self._listeners:
            listener.on_hosts_updated(timestamp, update_res)
```

3.2.4 通知 State

我们再梳理以下流程：

- 当Driver的定时进程通过节点发现脚本发现某一个节点被标记为新增或者移除时，它将发送一个通知到所有workers。
- 每一个 worker 有自己对应的 State，都被存储于 `WorkerNotificationManager . _listeners`。
- _host_messages 会在state 之中注册host的变化，就是往其 _host_messages 之中放入"host 有变化" 的消息。
- 因为这个消息不是一定要立即处理的，所以这里只是先放入 State 的队列之中。

逻辑如下：



3.2.5 何时处理

何时处理这个通知？在下一次 **state.commit()** 或者更轻量的 **state.check_host_updates()** 被调用时，state.check_host_updates 会从 _host_messages 中读取消息，积累更新。

如 check_host_updates 方法中注释所述，会在每个 worker 之间同步状态，目的是让这些 worker 同时抛出 **HostsUpdateInterrupt** 异常，具体同步使用 _bcast_object（然后内部调用到了 MPI）。

我们接下来就会在 State 的介绍之中，讲解check_host_updates 。

0x04 状态抽象

Horovod 实现了一个 State 对象，这是把机器训练的模型又做了一步抽象。

每一个Worker拥有一个 State 对象。

- Horovod 把所有需要在workers之间同步的变量都放进 hvd.elastic.State （比如model parameters，optimizer state，当前epoch和batch进度等等）对象之中。
- State 对象的作用是定期存储训练状态，在需要时候从 State 对象中恢复机器学习的状态。这样在某些worker发生意外错误时，可以避免因为状态被损坏而无法恢复现场。
- 比如，假设一个worker刚好在参数更新过程中突然挂掉，而此时部分梯度更新可能只更新到一半，这个状态是不可逆而又无法继续，导致参数是被损坏状态无法用于恢复训练。

4.1 State

State 的作用是：在不同的 worker 之中跟踪内存状态。

主要变量&方法是：

- on_reset : 当需要重启状态时候调用；
- on_hosts_updated： 当有 host 变化时候调用，即 向 _host_messages 这个 queue 放入一个消息；
- commit： 用户会定期调用此函数，会

  存储状态（state）到内存，检查 host 更改

  ；
  - 当有异常发生时，会抛出一个 HorovodInternalError 异常，当 hvd.elastic.run 捕获到这个异常后，会利用最新一次commit中恢复所有状态。
  - 因为commit状态代价高昂（比如如参数量太大会导致耗时过长），所以需要在"每个batch的处理时间"与"如果出错，训练需要从多久前的状态恢复"之间选取一个平衡点。比如，如果你每训练10个batches就commit一次，你就把复制时间降低了10倍。但是当发生错误时，你需要回滚到10个batches前的状态。
- check_host_updates : 会从

  ```
  _host_messages
  ```

  中读取消息，积累更新，如方法中注释所述，会在每个 worker 之间同步状态，目的是让这些 worker 同时抛出异常。具体同步使用

  ```
  _bcast_object
  ```

  （然后内部调用到了 MPI）；
  - 如果节点发现脚本可以预见到某个节点是需要被移除或新增，Elastic Horvod可以避免回滚操作。当Driver的定时进程通过节点发现脚本发现某一个节点被标记为新增或者移除时，它将发送一个通知到所有workers，于是在下一次 **state.commit()** 或者更轻量的 **state.check_host_updates()** 被调用时，会抛出一个 **HostsUpdateInterrupt** 异常。这个异常类似于 **HorovodInternalError** 异常，但是参数状态等不会从最近一次commit中恢复，而是从当前实时的参数中恢复。
  - 一般来说，如果你的硬件设施是可靠与稳定的，并且你的编排系统会在任务节点移除时提供足够的告警，你就可低频次调用 state.commit() 函数，同时只在每个batch结束时调用相对不耗时的 state.check_host_updates() 来检查节点变更情况。
- _reset_callbacks： 用户可以注册一些回调函数到

  hvd.elastic.State

  对象中，用于响应worker成员发生变化的情况。
  - 比如回调函数可以处理如下情况：
    1. 当worker数量发生改变时，学习率需要根据新的world size进行相应改变。

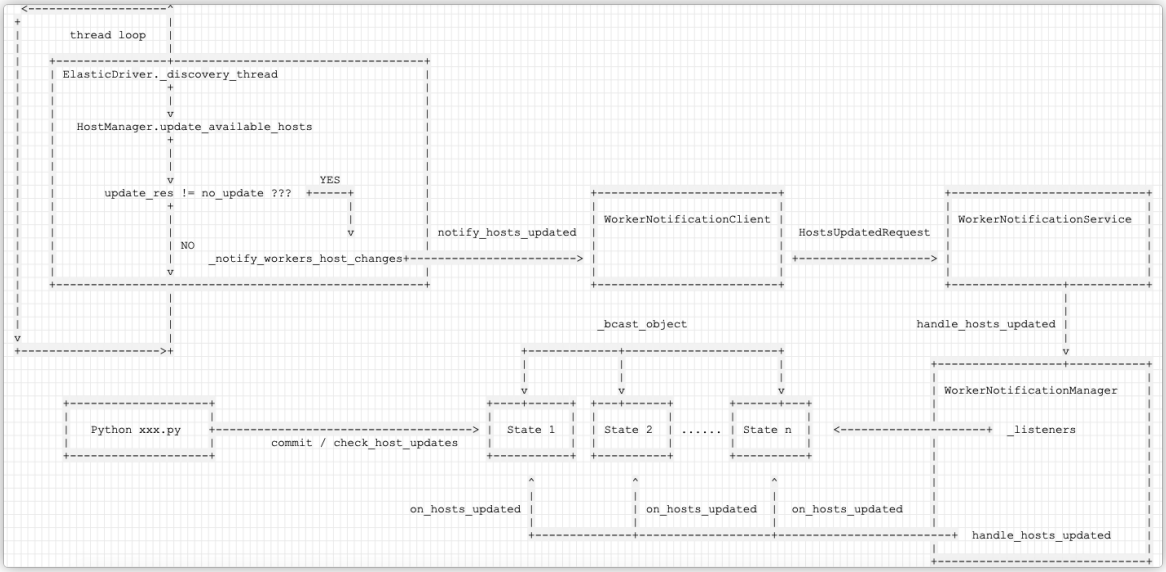
2. 对数据集进行重新分区。
- 这些回调函数会在"Horovod被重启之后"和"状态在节点间同步之前"这两个阶段中间被调用。

具体定义如下：

```
class State(object):
    """State representation used for tracking in memory state across workers.

    Args:
        bcast_object: Function used to broadcast a variable from rank 0 to the
other workers.
        get_rank: Function that returns the current rank of this worker.
    """
    def __init__(self, bcast_object, get_rank):
        self._bcast_object = bcast_object
        self._rank = get_rank
        self._host_messages = queue.Queue()
        self._last_updated_timestamp = 0
        self._reset_callbacks = []

    def on_reset(self):
        self._host_messages = queue.Queue()
        self.reset()
        for callback in self._reset_callbacks:
            callback()

    def on_hosts_updated(self, timestamp, update_res):
        self._host_messages.put((timestamp, update_res))

    def commit(self):
        self.save()
        self.check_host_updates()

    def check_host_updates(self):
        """Checks that a notification has been sent indicating that hosts can be
added or will be removed.

        Raises a `HostsUpdatedInterrupt` if such a notification has been
received.
        """
        # Iterate through the update messages sent from the server. If the
update timestamp
        # is greater than the last update timestamp, then trigger a
HostsUpdatedException.
        # 遍历更新消息，如果更新时间戳大于上次更新时间戳，就触发一个HostUpdateResult
        last_updated_timestamp = prev_timestamp = self._last_updated_timestamp
        all_update = HostUpdateResult.no_update
        while not self._host_messages.empty():
            timestamp, update = self._host_messages.get()
            if timestamp > last_updated_timestamp:
                last_updated_timestamp = timestamp
                all_update |= update

        # In order to ensure all workers raise the exception at the same time,
we need to sync
        # the updated state across all the workers.
```

```
        # TODO(travis): this should be a max allreduce to account for changes in
rank 0
        # 会从 `_host_messages` 中读取消息，积累更新，如方法中注释所述，会在每个 worker 之
间同步状态，目的是让这些 worker 同时抛出异常。具体同步使用 `_bcast_object`（然后内部调用到了
MPI）
        prev_timestamp, self._last_updated_timestamp, all_update = \
            self._bcast_object((prev_timestamp, last_updated_timestamp,
all_update))

        # At this point, updated state is globally consistent across all ranks.
        if self._last_updated_timestamp > prev_timestamp:
            raise HostsUpdatedInterrupt(all_update == HostUpdateResult.removed)
```

因此，我们加入 Commit 之后，逻辑如图：



我们接下来介绍各级派生类。

4.2 ObjectState

ObjectState 的目的是组装成 simple Python objects。

```
class ObjectState(State):
    """State for simple Python objects.

    Every object is specified as a keyword argument, and will be assigned as an
attribute.

    Args:
        bcast_object: Horovod broadcast object function used to sync state
dictionary.
        get_rank: Horovod rank function used to identify is this process is the
coordinator.
        kwargs: Properties to sync, will be exposed as attributes of the object.
    """
    def __init__(self, bcast_object, get_rank, **kwargs):
        self._bcast_object = bcast_object
        self._saved_state = kwargs
        self._set_attrs()
        super(ObjectState, self).__init__(bcast_object=bcast_object,
get_rank=get_rank)
```

```
    def save(self):
        new_state = {}
        for attr in self._saved_state.keys():
            new_state[attr] = getattr(self, attr)
        self._saved_state = new_state

    def restore(self):
        self._set_attrs()

    def sync(self):
        if self._saved_state:
            self._saved_state = self._bcast_object(self._saved_state)
            self._set_attrs()

    def _set_attrs(self):
        for attr, value in self._saved_state.items():
            setattr(self, attr, value)
```

4.3 TensorFlowKerasState

Horovod 默认已提供标准的TensorFlow，Keras和PyTorch的状态保持和恢复实现，如果需要在某些场景下自定义，可以重载 **hvd.elastic.State** 这个对象。

TensorFlowKerasState 是 TensorFlow Keras model and optimizer 的状态抽象。

初始化函数中，会设置各种相关变量，比如广播函数。

```
class TensorFlowKerasState(ObjectState):

    def __init__(self, model, optimizer=None, backend=None, **kwargs):
        self.model = model
        if not _model_built(model):
            raise ValueError('Model must be built first. Run
`model.build(input_shape)`.')

        self.optimizer = optimizer or model.optimizer
        self.backend = backend
        self._save_model()

        if not backend or _executing_eagerly():
            self._bcast_model = lambda: _broadcast_model(self.model,
self.optimizer, backend=self.backend)
            bcast_object = broadcast_object
        else:
            # For TensorFlow v1, we need to reuse the broadcast op to prevent
incrementing the uids
            bcast_op = broadcast_variables(_global_variables(), root_rank=0)
            self._bcast_model = lambda: self.backend.get_session().run(bcast_op)
            bcast_object =
broadcast_object_fn(session=self.backend.get_session())

        super(TensorFlowKerasState, self).__init__(bcast_object=bcast_object,
                                                   get_rank=rank,
                                                   **kwargs)
```

具体实现了几个方法，基本就是 存储，恢复 state，同步。

```
def save(self):
    self._save_model()
    super(TensorFlowKerasState, self).save()

def restore(self):
    self._load_model()
    super(TensorFlowKerasState, self).restore()

def sync(self):
    self._bcast_model()
    self._save_model()
    super(TensorFlowKerasState, self).sync()

def _save_model(self):
    if _executing_eagerly():
        self._saved_model_state = [tf.identity(var) for var in
self.model.variables]
        self._saved_optimizer_state = [tf.identity(var) for var in
self.optimizer.variables()]
    else:
        self._saved_model_state = self.model.get_weights()
        self._saved_optimizer_state = self.optimizer.get_weights()

def _load_model(self):
    if _executing_eagerly():
        for var, saved_var in zip(self.model.variables,
self._saved_model_state):
            var.assign(saved_var)
        for var, saved_var in zip(self.optimizer.variables(),
self._saved_optimizer_state):
            var.assign(saved_var)
    else:
        self.model.set_weights(self._saved_model_state)
        self.optimizer.set_weights(self._saved_optimizer_state)
```

4.4 Restore

我们看到了，restore 会从内存中恢复模型。

```
def restore(self):
    self._load_model()
    super(TensorFlowKerasState, self).restore()
```

于是，我们有一个问题：何时调用restore?

发现是如果 horovod 捕获了 HorovodInternalError 之后，会用 restore 来恢复。

```
def run_fn(func, reset):
    @functools.wraps(func)
    def wrapper(state, *args, **kwargs):
        notification_manager.init()
        notification_manager.register_listener(state)
        skip_sync = False

        try:
            while True:
```

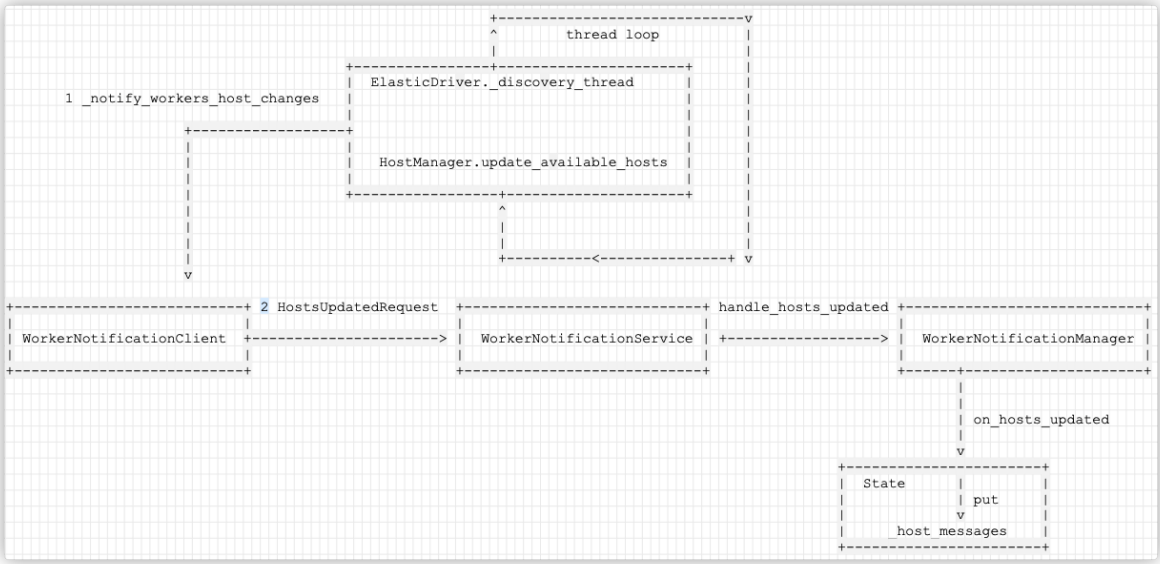
```
                if not skip_sync:
                    state.sync()

                try:
                    return func(state, *args, **kwargs)
                except HorovodInternalError:
                    state.restore() # 在这里调用
                    skip_sync = False
                except HostsUpdatedInterrupt as e:
                    skip_sync = e.skip_sync

                reset()
                state.on_reset()
        finally:
            notification_manager.remove_listener(state)
    return wrapper
```

0x05 总结

我们再次重复一下，发现节点机制的几个关键设计点：

- **有节点变化时候，如何即时发现**？Horovod是通过定期调用完成。
- **发现节点变化时候，如何通知各个worker**？Horovod通过构建了一个通知机制完成。即，每个worker把自己注册到WorkerNotificationManager 之上，当有节点变化时候，WorkerNotificationManager 会逐一通知这些worker。
- **worker得到通知之后，如何处理**？Horovod 把worker的状态在深度框架上进一步封装成各种State，得到通知之后就会调用State的对应callback函数，或者同步状态，或者进行其他处理。

简化版总体逻辑如下：



至此，发现节点部分介绍完毕，因为本文只是使用了 WorkerNotificationService 完成通知，但是没有深入介绍，所以下一篇介绍内部广播和通知机制。