horovod (15) --- 广播 & 通知

目录

0x00 摘要

看看horovod 弹性训练如何广播和发送通知。

0x01 问题

首先，我们提出一个问题：为什么弹性训练 需要有广播?

答案就是：因为捕获两种异常之后，需要广播到各个worker。

1.1 HorovodInternalError

关于 HorovodInternalError 异常处理，我们看看具体容错机制，就可以知道缘由：

- 在 `hvd.elastic.run` 装饰器捕获异常；
- 如果是 `HorovodInternalError`，就恢复到最近一次提交的状态，此时因为是allreduce等异常，所以所有worker都处于停止状态；
- driver 会根据当前正在运行的节点重新执行一个 rendezvous，以便重新初始化 Horovod context；
- 当新的通信域构造成功后，rank = 0 的 worker 会将自身的模型广播给其他 worker；
- 所有worker接着上次停止的迭代步数继续训练；

因为需要从 rank 0 广播变量给其他进程，所以必须有一个广播机制。

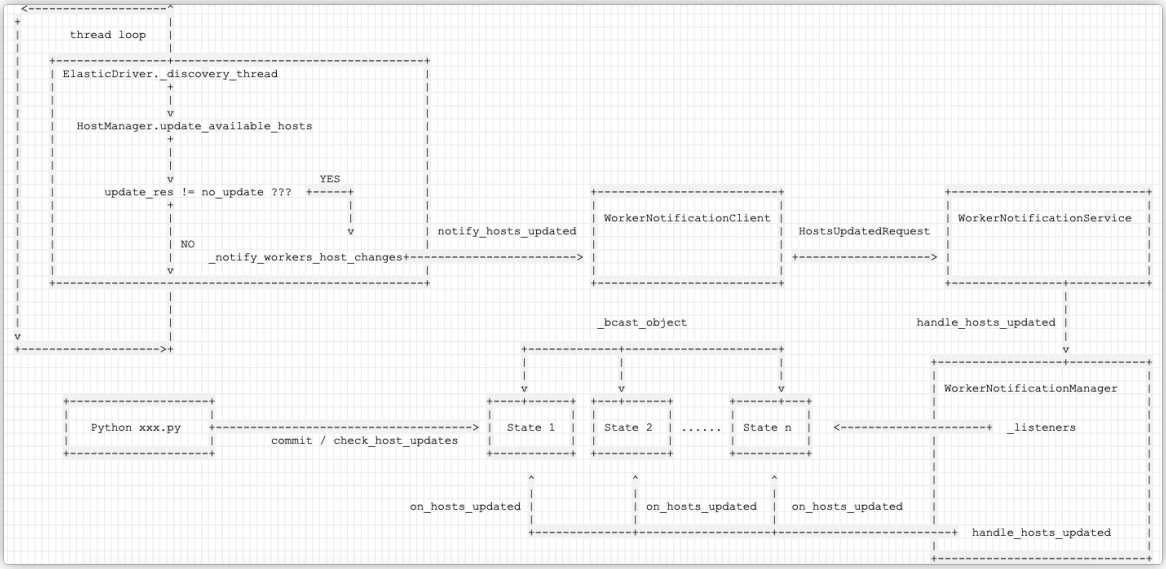
1.2 HostsUpdateInterrupt

关于 HostsUpdateInterrupt 异常处理，我们看看具体原因。

- 当驱动进程通过节点发现脚本发现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，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。
- check_host_updates 方法 会从 `_host_messages` 中读取消息，积累更新，如其方法中注释所述，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）。

需要一个广播机制在每个 worker 之间同步状态（因为这些worker目前都是在正常训练，需要有一个东西统一打断他们的训练，从而重新组建一个通信环），目的是让这些 worker 同时抛出 HostsUpdateInterrupt 异常。

其次，我们需要回顾下上文的流程图，本文将对其部分内部流程进行细化。



0x02 广播机制

我们具体剖析广播机制如下，因为广播是和具体框架密切结合，所以我们以tensorflow为例，具体代码在horovod/tensorflow/elastic.py 之中。

2.1 广播实现

在 horovod/tensorflow/elastic.py 之中，就是针对 TF 做的特定实现。其中会依据 TF 的版本做不同处理。

2.1.1 TensorFlowKerasState

以 TensorFlowKerasState 为例，在初始化的时候，因为有广播对象的需要，比如在 TensorFlowKerasState 之中配置了`_bcast_model`用来广播模型，`bcast_object`用来广播对象，`broadcast_variables`用来广播变量。

而且提供了sync函数负责广播，可以看出来调用了`_bcast_model`。

```
class TensorFlowKerasState(ObjectState):
    def __init__(self, model, optimizer=None, backend=None, **kwargs):

        if not backend or _executing_eagerly():
            # 这里设定了广播函数
            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
            # 这里设定了广播函数
            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())

    def sync(self):
        self._bcast_model() #广播模型
        self._save_model()
        super(TensorFlowKerasState, self).sync()
```

2.1.2 广播模型

_broadcast_model 函数会 广播 模型变量，optimizer变量。

```
def _broadcast_model(model, optimizer, backend):
    if _executing_eagerly():
        # TensorFlow 2.0 or TensorFlow eager
        broadcast_variables(model.variables, root_rank=0) # 广播模型变量
        broadcast_variables(optimizer.variables(), root_rank=0) # 广播优化器变量
    else:
        bcast_op = broadcast_variables(_global_variables(), root_rank=0)
        backend.get_session().run(bcast_op)
```

2.1.3 广播变量

广播变量的具体实现 在 horovod/tensorflow/functions.py 之中。broadcast_variables 的作用是从 root rank（即 rank 0）广播变量到其他的进程。

具体也根据 TF 版本做了区别。

```
def _make_subgraph(f):
```

```
    return tf.function(f)

@_cache
def _make_broadcast_group_fn():
    if _executing_eagerly():
        # Eager mode will parallelize independent control flow
        def broadcast_group(variables, root_rank): # 在这里定义
            for var in variables:
                var.assign(broadcast(var, root_rank)) # 调用MPI函数，这里都指定了是
root_rank

        return _make_subgraph(broadcast_group)
    else:
        # Graph mode requires an Op
        def broadcast_group(variables, root_rank): # 在这里定义
            # tf.group()用于创造一个操作，可以将传入参数的所有操作组合，当这个操作完成后,所
有 input 中的所有 ops 都已完成。tf.group()操作没有输出。
            return tf.group(*[var.assign(broadcast(var, root_rank)) # 这里调用MPI
函数
                              for var in variables])

        return broadcast_group

def broadcast_variables(variables, root_rank):
    """Broadcasts variables from root rank to all other processes.
    """
    broadcast_group = _make_broadcast_group_fn()
    return broadcast_group(variables, root_rank # 在上面定义
```

2.1.4 广播对象

广播对象 的作用是从 root rank（即 rank 0）广播对象到其他的进程。 广播对象和广播变量的区别是：对象需要序列化和反序列化。

```
def broadcast_object(obj, root_rank=0, session=None, name=None):
    """
    Serializes and broadcasts an object from root rank to all other processes.

    Arguments:
        obj: An object capable of being serialized without losing any context.
        root_rank: The rank of the process from which parameters will be
                   broadcasted to all other processes.
        session: Session for TensorFlow v1 compatibility.
        name: Optional name to use during broadcast, will default to the class
              type.
    Returns:
        The object that was broadcast from the `root_rank`.
    """
    if name is None:
        name = type(obj).__name__

    def to_numpy(v): # 依据tf版本不同做不同处理
        if not _executing_eagerly():
            sess = session or ops.get_default_session()
            return sess.run(v)
        else:
            return v.numpy()
```

```
    if rank() == root_rank:
        b = io.BytesIO() # BytesIO实现了在内存中读写bytes
        cloudpickle.dump(obj, b) # 序列化，编码成一个二进制文件
        t = tf.convert_to_tensor(bytearray(b.getvalue()), dtype=tf.uint8)
        sz = tf.convert_to_tensor([t.shape[0]], dtype=tf.int32) # 张量对应维度的数值
        to_numpy(broadcast(sz, root_rank, name + '.sz')) # 广播维度
    else:
        sz = tf.convert_to_tensor([0], dtype=tf.int32)
        sz = to_numpy(broadcast(sz, root_rank, name + '.sz')) # 接受维度
        t = tf.zeros(sz.tolist()[0], dtype=tf.uint8)

    t = to_numpy(broadcast(t, root_rank, name + '.t')) # 广播对象内容

    if rank() != root_rank:
        buf = io.BytesIO(t.tobytes())
        obj = cloudpickle.load(buf) # 反序列化，解码成原本的对象

    return obj
```

2.1.5 HVD C++

底层会调用到 MPI 函数完成广播功能。

```
def broadcast(tensor, root_rank, name=None, ignore_name_scope=False):
    """An op which broadcasts the input tensor on root rank to the same input
tensor
    on all other Horovod processes.

    The broadcast operation is keyed by the name of the op. The tensor type and
    shape must be the same on all Horovod processes for a given name. The
broadcast
    will not start until all processes are ready to send and receive the tensor.

    Returns:
      A tensor of the same shape and type as `tensor`, with the value
broadcasted
      from root rank.
    """
    if name is None and not _executing_eagerly():
        name = 'HorovodBroadcast_%s' % _normalize_name(tensor.name)
    return MPI_LIB.horovod_broadcast(tensor, name=name, root_rank=root_rank,
                                     ignore_name_scope=ignore_name_scope)
```

2.1.6 MPI

MPI_BCAST的作用是：从一个序列号为root的进程将一条消息广播发送到组内的所有进程, 包括它本身在内。

因为之前指定了root_rank，所以即使所有worker虽然都调用了同样代码，也只是会把 root_rank 通信消息缓冲区中的消息拷贝到其他所有进程中去。

```
void MPIController::Bcast(void* buffer, size_t size, int root_rank,
                          Communicator communicator) {
  MPI_Comm comm = mpi_ctx_.GetMPICommunicator(communicator);
  int ret_code = MPI_Bcast(buffer, size, MPI_BYTE, root_rank, comm);
  if (ret_code != MPI_SUCCESS) {
    throw std::runtime_error(
        "MPI_Broadcast failed, see MPI output for details.");
  }
}
```

2.1.7 小结

我们总结一下各个函数：

- `_bcast_model` 用来广播模型；
- `bcast_object` 用来广播对象；
- `broadcast_variables` 用来广播变量；
- 广播对象和广播变量的区别是：对象需要序列化和反序列化。
- `_broadcast_model` 就是调用了 `broadcast_variables` 完成对模型参数的广播；
- `broadcast_variables` 中调用了 `broadcast_group`，`broadcast_group` 主要就是利用 tf.group()把广播操作组合起来；

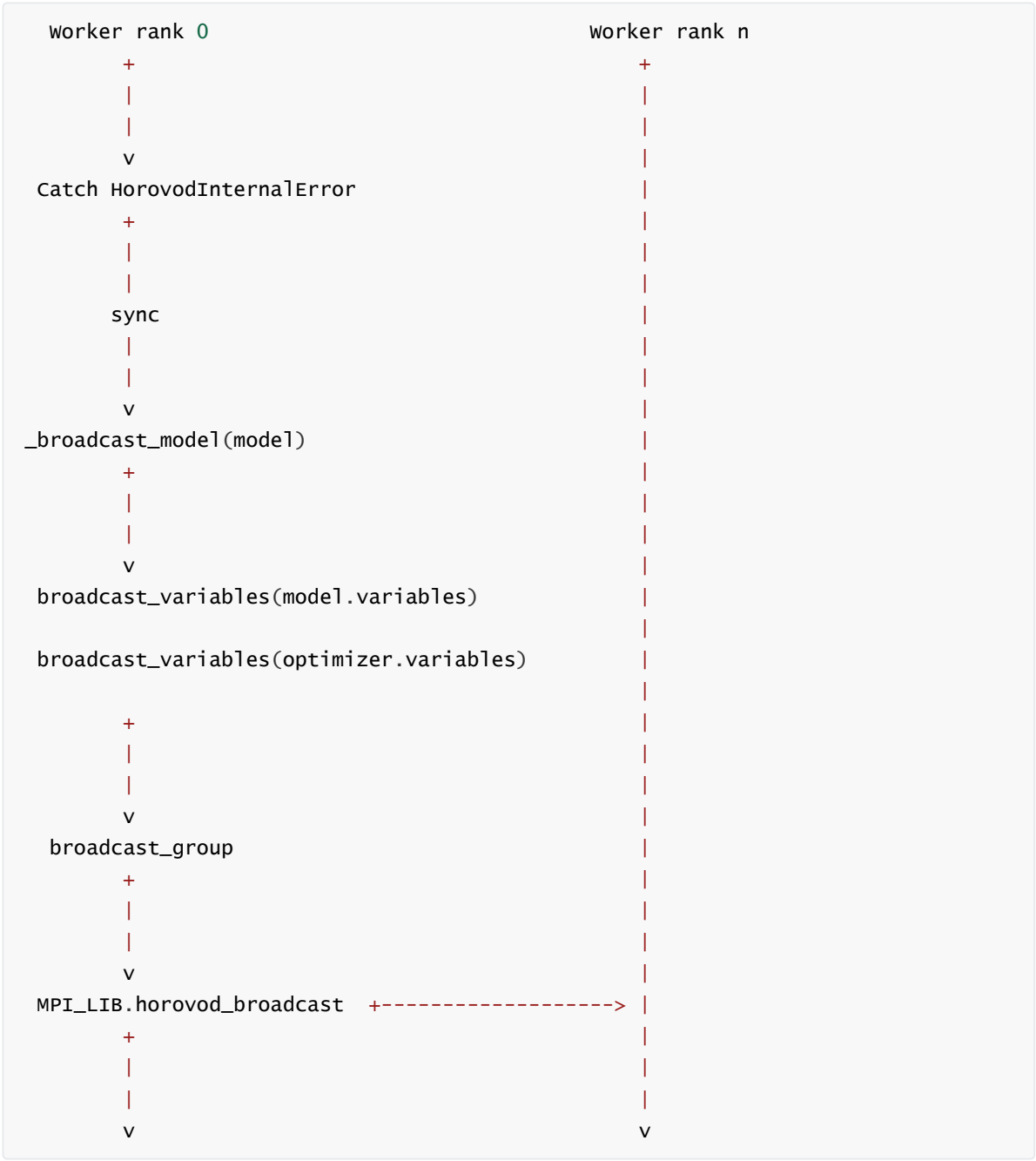
2.2 使用

2.2.1 HorovodInternalError

当捕获HorovodInternalError时候，会进行广播同步，目的是当新的通信域构造成功后，rank = 0 的 worker 会将自身的模型广播给其他 worker。

```
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() # 这里会进行广播同步，就是TensorFlowKerasState.sync

                try:
                    return func(state, *args, **kwargs)
                except HorovodInternalError:
                    state.restore() # 捕获一场，然后继续while循环
                    skip_sync = False
                except HostsUpdatedInterrupt as e:
                    skip_sync = e.skip_sync

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

具体如下：



2.2.2 HostsUpdateInterrupt

广播对象作用是 在每个 worker 之间同步状态，目的是让这些 worker 同时抛出 HostsUpdateInterrupt 异常。

具体如何使用?

在 `workerNotificationService . _handle` 方法之中，调用了 `self._manager.handle_hosts_updated(req.timestamp, req.res)` 进行通知更新。

WorkerNotificationManager. handle_hosts_updated 方法之中，会调用注册的 state，逐一通知更新。

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

是在 State 的几个方法中可以看到。

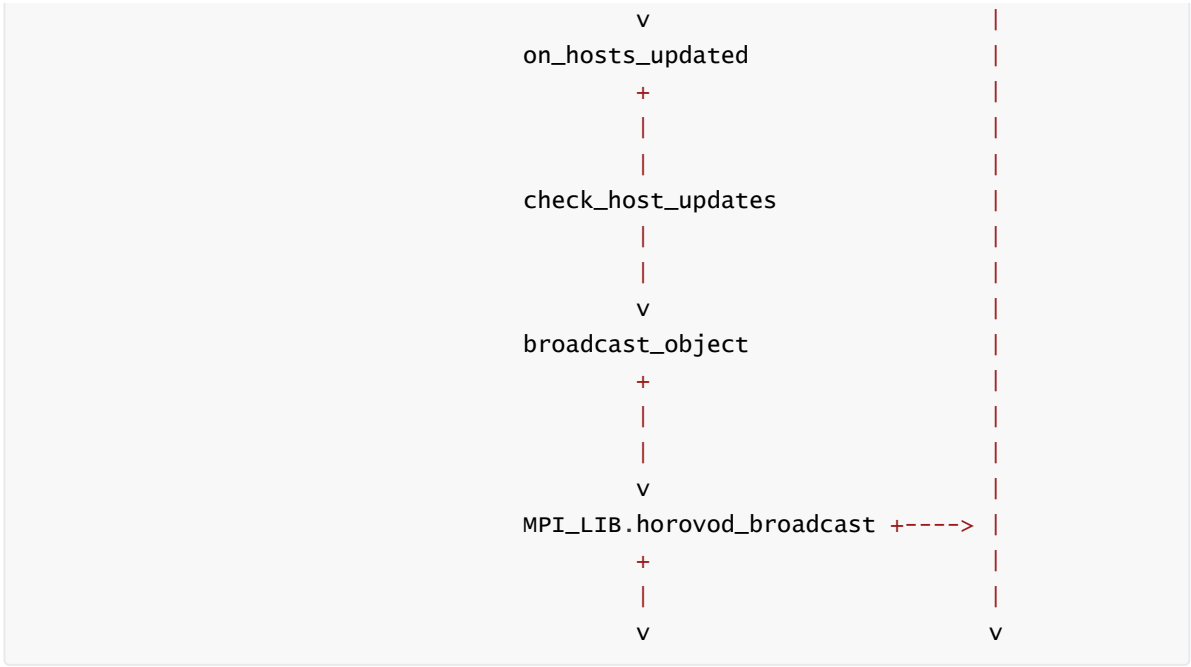
- on_hosts_updated： 当有 host 变化时候调用，即 向 `_host_messages` 这个 queue 放入一个消息；
- commit： 用户会定期调用此函数，会存储状态，检查 host 更改；
- check_host_updates : 会从 `_host_messages` 中读取消息，积累更新，如方法中注释所述，会在每个 worker 之间同步状态，目的是让这些 worker 同时抛出异常。具体同步使用 `_bcast_object`；

check_host_updates代码如下：

```
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.
    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
    # 这里会广播
    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)
```

具体如下：

```
+---------------------------+      +--------------+           +-------------+
|Catch HostsUpdatedInterrupt|      | Worker rank 1|           |Worker rank n|
+---------+-----------------+      +-------+------+           +-----+-------+
          |                              |                        |
          |                              |                        |
          v                              |                        |
 WorkerNotificationService               |                        |
          +                              |                        |
          |                              |                        |
          |                              |                        |
          v                              |                        |
manager.handle_hosts_updated+----------> |                        |
                                         |                        |
                                         |                        |
```



0x03 通知机制

上图中用到 manager.handle_hosts_updated，manager就是 WorkerNotificationManager。

所以我们顺着讨论下 WorkerNotificationManager，这是 Hovorod 的通知机制。

3.1 WorkerNotificationManager 生成

每个host 只有一个 WorkerNotificationManager，也只有一个 WorkerNotificationService。

注意：是 ElasticDriver 会作为client，给这些 WorkerNotificationService 发消息，从而引起 WorkerNotificationManager 的对应操作。

horovod/common/elastic.py 有如下代码完成了实例生成。

```
notification_manager = WorkerNotificationManager()
```

WorkerNotificationManager 定义如下：

```
class WorkerNotificationManager(object):
    def __init__(self):
        self._lock = threading.Lock()
        self._service = WorkerNotificationService(secret_key, nic, self)
        self._listeners = set()
```

3.2 初始化

在用户代码启动之前，会先初始化 WorkerNotificationManager。

```
 def run_fn(func, reset):
    @functools.wraps(func)
    def wrapper(state, *args, **kwargs):
        # 初始化 WorkerNotificationManager
        notification_manager.init()
        # 把自己对应的 state 注册到 notification_manager
        notification_manager.register_listener(state)
```

WorkerNotificationManager初始化代码如下，其逻辑是：

- 如果 _service 已经生成，则直接返回，这就保证了每个host之中只有一个 WorkerNotificationService。
- 从系统变量中得到 rendezvous 的各种信息，比如地址，端口，key 等等；
- 生成 WorkerNotificationService，赋值给 _service；
- 使用 put_data_into_kvstore 把 本 worker 的地址 和 给其在逻辑通信环分配的序号 rank 发送给 rendezvous（这个为了后续生成 WorkerNotificationClient 使用）。
- 备注：这个 rendezvous 会存储每个 worker 的地址和给其在逻辑通信环分配的序号 rank。worker 进程可以通过这个 rendezvous 来构造新的通信域。

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

        # 从系统变量中得到 rendezvous 的各种信息，比如地址，端口，key 等等
        rendezvous_addr = rendezvous_addr or
os.environ.get(HOROVOD_GLOO_RENDEZVOUS_ADDR)
        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)
        # 把本worker的地址 和 给其在逻辑通信环分配的序号 rank 发送给 rendezvous
        put_data_into_kvstore(rendezvous_addr,
                              rendezvous_port,
                              PUT_WORKER_ADDRESSES,
                              self._create_id(hostname, local_rank),
                              value)
```

具体 put_data_into_kvstore 如下。

```
def put_data_into_kvstore(addr, port, scope, key, value):
    try:
        url = "http://{addr}:{port}/{scope}/{key}".format(
            addr=addr, port=str(port), scope=scope, key=key
        )
        req = Request(url, data=codec.dumps_base64(value, to_ascii=False))
        req.get_method = lambda: "PUT"  # for urllib2 compatibility
        urlopen(req)
    except (HTTPError, URLError) as e:
        raise RuntimeError("Put data input KVStore server failed.", e)
```

3.3 注册State

用户代码启动之前，还会把自己对应的 state 注册到 notification_manager。

```
def run_fn(func, reset):
    @functools.wraps(func)
    def wrapper(state, *args, **kwargs):
        # 初始化 WorkerNotificationManager
        notification_manager.init()
        # 把自己对应的 state 注册到 notification_manager
        notification_manager.register_listener(state)
```

具体代码如下：

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

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

3.4 WorkerNotificationService

WorkerNotificationService 在每个host之中也只有一个，用来接受其 client 发来的 HostsUpdatedRequest 消息，进行处理。可以看到，其继承了 network.BasicService，这意味着 WorkerNotificationService 本身是一个http server，可以和其client交互，大家可以想想之前介绍的各种 driver / client，就可以理解其机制了。

```
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)
```

逻辑如下：

该图无法显示，看网页

3.5 WorkerNotificationClient

WorkerNotificationClient 就是用来给 WorkerNotificationService 发送消息的接口。

ElasticDriver 中，会针对每个 worker 生成一个对应的 WorkerNotificationClient，用来进行通知。

```
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))
```

3.6 生成 Client

3.6.1 注册时机

回顾一下，在 WorkerNotificationManager 的初始化函数 init 中，会给 rendezvous 发送put 请求，进行注册。

注册信息就是为了 生成client。

```
put_data_into_kvstore(rendezvous_addr,
                      rendezvous_port,
                      PUT_WORKER_ADDRESSES,
                      self._create_id(hostname, local_rank),
                      value)
```

3.6.2 注册 worker

在 ElasticRendezvousHandler 中有 _put_value，用来处理 PUT_WORKER_ADDRESSES。调用 driver 处理。

```
# 注意，这里在 Rendezvous Server 之内
def _put_value(self, scope, key, value):
    if scope == PUT_WORKER_ADDRESSES:
        host, local_rank = key.split(':')
        addresses, secret_key = codec.loads_base64(value)
        self._put_worker_addresses(host, int(local_rank), addresses, secret_key)

    super(RendezvousHandler, self)._put_value(scope, key, value)

def _put_worker_addresses(self, host, local_rank, addresses, secret_key):
    # 这里调用driver进行处理
    driver.register_worker_server(host, local_rank, addresses, secret_key)
```

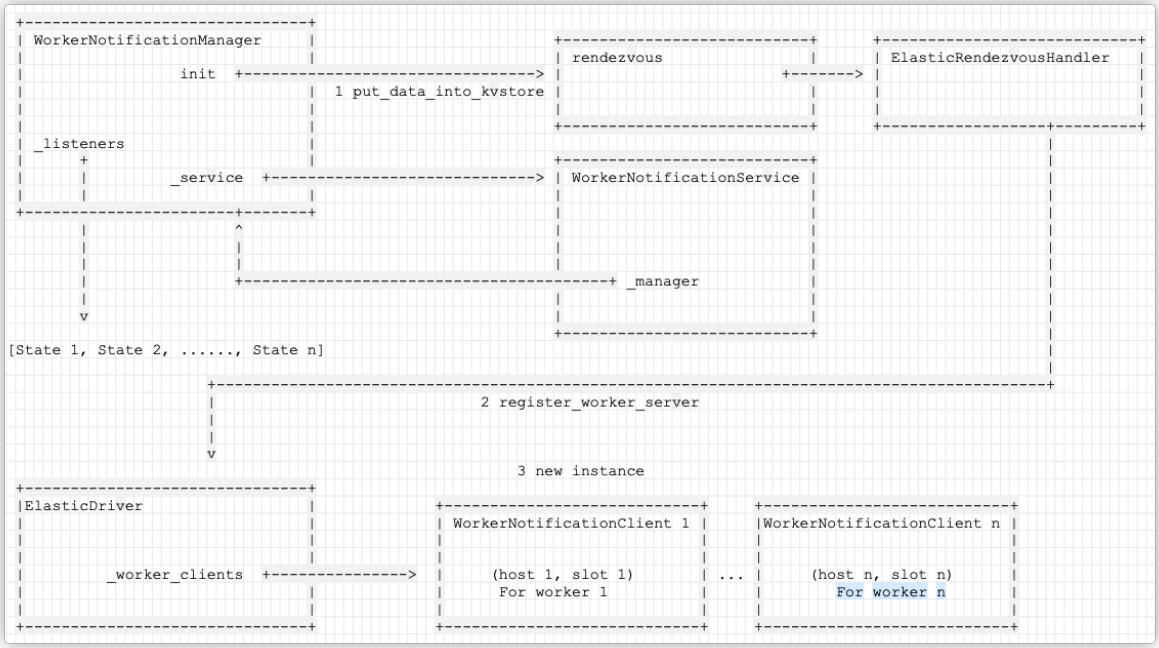
3.6.3 生成 WorkerNotificationClient

ElasticDriver 中，会针对每个 worker 生成一个对应的 WorkerNotificationClient，用来进行通知。

这里需要注意：ElasticDriver 就是 WorkerNotificationClient 的使用者，需要通知各个worker时候，就调用这些WorkerNotificationClient ，给对应host上的 WorkerNotificationService发消息，从而引起WorkerNotificationManager做相应处理。

```
# 这里是 ElasticDriver 之中
def register_worker_server(self, host, slot, addresses, secret_key):
    self._worker_clients[(host, slot)] = WorkerNotificationClient(
        addresses, secret_key, self._verbose)
```

逻辑如下：



3.7 使用

3.7.1 发现更新

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

```
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()
    coordinator_client = self.get_worker_client(coordinator_slot_info)

    timestamp = _epoch_time_s()
```

```
    coordinator_client.notify_hosts_updated(timestamp, update_res)
```

3.7.2 获取 client

get_worker_client 函数就是获取 WorkerNotificationClient。就是依据 host，slot 信息来找到某一个 worker 对应的 client。

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

3.7.3 发送HostsUpdatedRequest

notify_hosts_updated 的作用是发送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))
```

3.7.4 处理 HostsUpdatedRequest

WorkerNotificationService 之中会处理HostsUpdatedRequest，调用 WorkerNotificationManager处理。

```
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)
```

3.7.5 WorkerNotificationManager

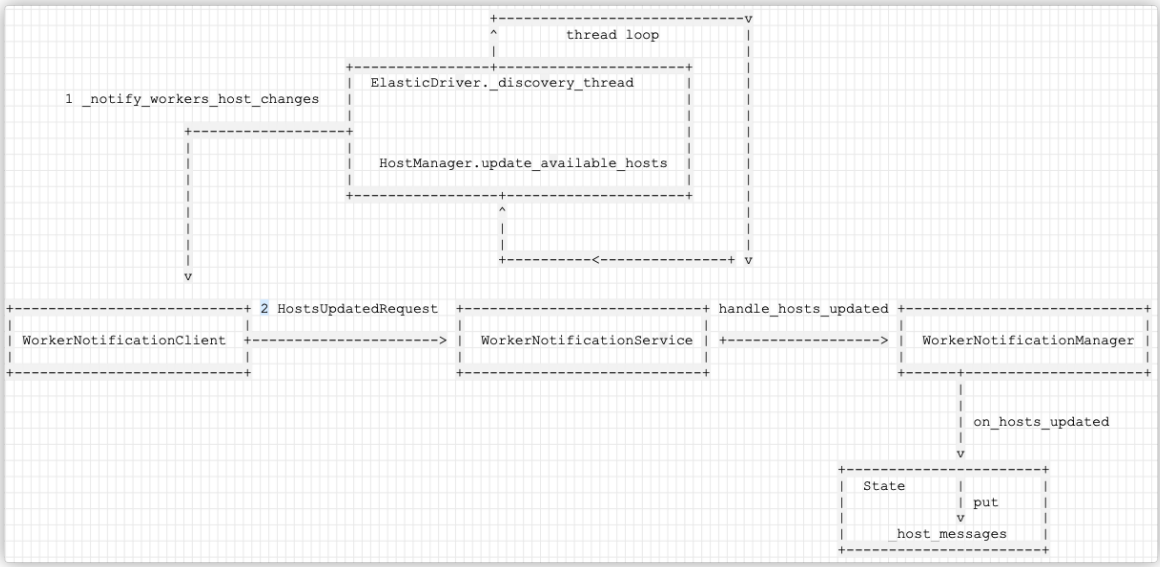
所以，当有host 更新时候，WorkerNotificationManager 中的 handle_hosts_updated 如下，最终调用到 state 的 on_hosts_updated。

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

State 的实现如下：

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

逻辑如下图：



3.7.6 处理更新

在用户调用 commit 的时候，才会调用 check_host_updates 检查更新。

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

检查更新就是看看 _host_messages 有没有新的消息，如果发现 host 有变化，就会产生一个 HostsUpdatedInterrupt 异常。

```
def check_host_updates(self):
    # Iterate through the update messages sent from the server. If the update
timestamp
    # is greater than the last update timestamp, then trigger a
HostsUpdatedException.
    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
    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)
```

在worker进程出现 **HorvodInternalError** 错误或者 **HostsUpdatedInterrupt** 节点增删时，会捕获这两个错误，调用 reset 来进行容错处理。于是就把流程前后串联了起来。

具体如下：