弹性训练总体架构

目录

0x01 总述

1.1 问题点

我们思考下，Horovod 目前遇到了什么问题？

- 无法自动调节容量（Auto Scale）
  - 因为计算资源也许会有弹性调度，所以应该考虑到如果集群缩容了怎么办？如果扩容了怎么？理想状态应该是：在训练过程中可以自动增加或者减少worker数量。而且在worker数量变化时，不会中断训练任务，做到平滑过渡。
  - 目前Horovod无法在资源有限的情况下执行。假如一共需要100个GPU，暂时只有40个GPU到位，在这种情况下，Horovod就只能等待，不能用现有的40个GPU先在少量进程上开始训练，从而无法快速开始模型迭代。
  - 资源充裕时，Horovod 无法自动增加进程加速训练。就上例而言，在理想状态下，Horovoid应该先用这40个GPU构建一个环来启动训练，如果发现60个新GPU到位了就自动动态扩容，从而在下一个 epoch 开始就用100个GPU构建新的环开始训练；
- **没有容错机制（Fault Tolerance）**。目前如果某一个节点失败，整个训练会失败，用户只能从头开始训练。如果可以支持 auto scale，加上一些之前陆续保存的 checkpoint，则Horovod可以重新选取一个好节点启动这个worker，或者用剩下的节点构建一个环继续训练。
- 调度机制不灵活
  - 机器学习训练任务一般时间较长，占用算力大，而Horovod任务缺少弹性能力，不支持动态配置 worker，不支持高优先级抢占实例。因此当资源不足时，无法按需为其他高优先级业务腾出资源， 只能等待任务自己主动终止或者出错终止。

为了解决以上几个问题，我们会思考很多的其他具体技术问题和细节，让我们先罗列出来：

- 何时构建 checkpoint？哪一个阶段是合适的？每一个 epoch 之后自动保存？还是由用户自行控制（这样可以做到更好的）？

- 如何从 checkpoint恢复?
- checkpoint需要存储哪些东西，即，对于horovod来说，哪些状态是必须的?
- 如何监听 worker 的工作情况？怎么判断机器出了问题？假如只是网络阻塞偶尔导致的怎么办?
- 需要构建一个通知机制；
- 如何知道集群的富余资源？如何发现可用节点?
- 如何构建新的通信环 ring?
- 如果构建新ring，是由一个 master 完成？还是使用类似 gossip 这样的协议?
- 是否有优先级调度，这样可以充分利用共享集群资源空闲的资源。
- 新 worker 怎么被 sync?
- 原有的active worker 节点怎么处理?
- 出问题的 worker 节点怎么处理?
- rank 0 怎么广播?

我们在本文以及后续各篇的分析中试着解答这些问题。

注：Horovod目前的调度机制依然不灵活，不支持抢占。

1.1 角色

Horovod 在单机的多个 GPU 之上采用 NCCL 来通信，在多机（CPU或者GPU）之间通过 Ring AllReduce 算法进行通信。Horovod 的弹性训练是指多机的弹性训练。

Horovod 弹性训练有两个角色：driver和 worker。driver 进程运行在 CPU 节点上，worker 进程可以运行在 CPU 节点或者 GPU 节点之上。

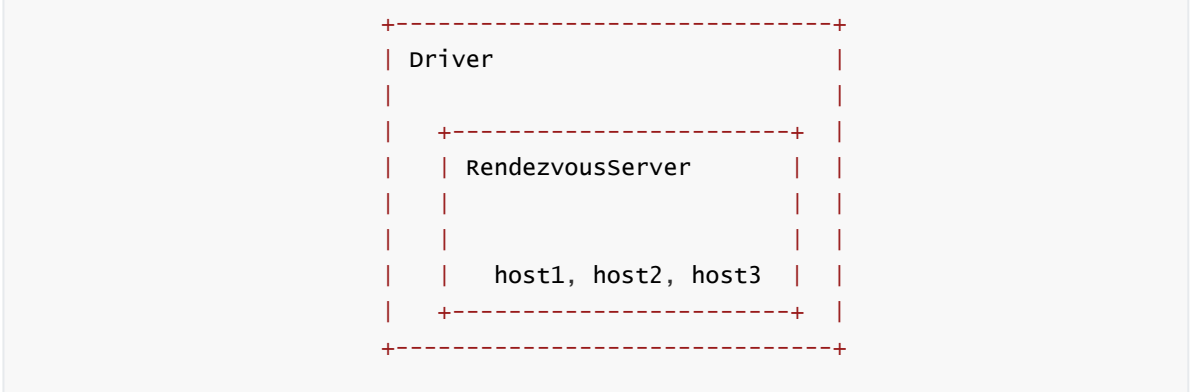
Driver 进程的作用是：

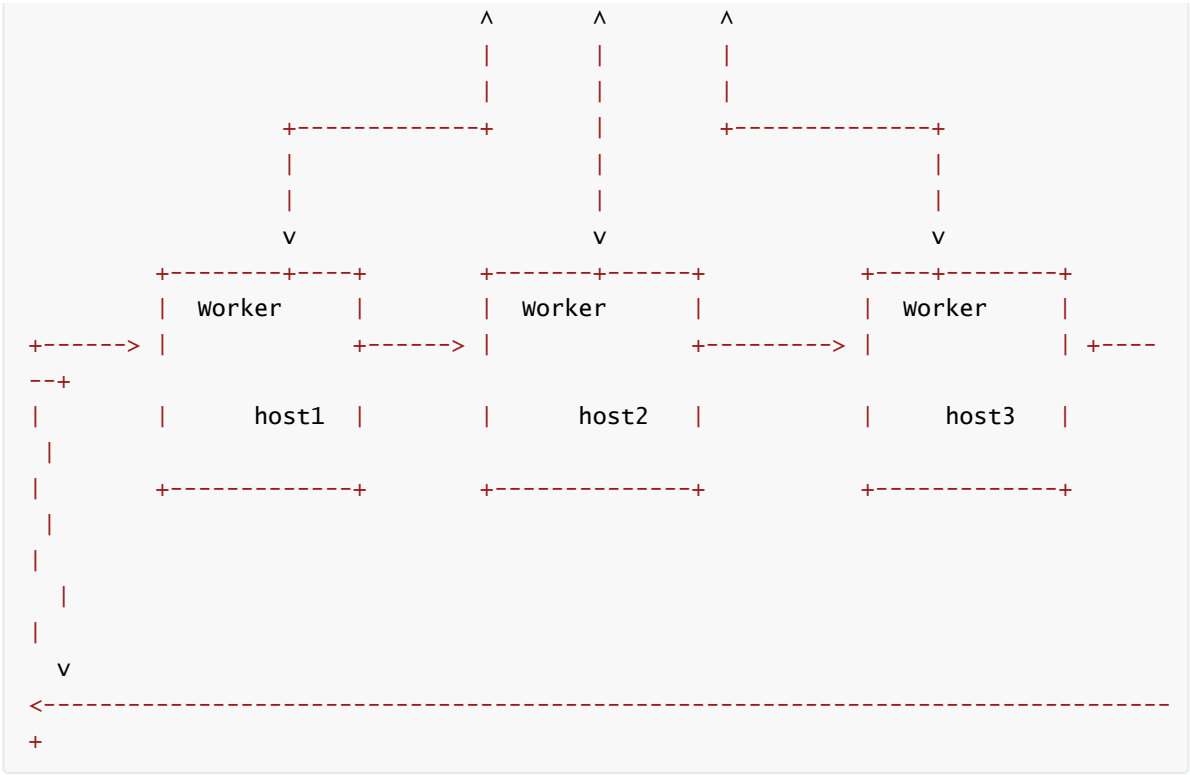
- 调用 Gloo 帮助 workers 构造一个 AllReduce 通信环，或者说是通信域。Driver 不参与具体构建通信环，而是提供辅助信息，从而worker可以建立环。
    - Driver 进程需要给 Gloo 创建一个带有 KVStore 的 RendezvousServer，其中 KVStore 用于存储通信域内每个节点的 host 和 其在逻辑通信环分配的序号 rank 等信息。
    - 这个 RendezvousServer 运行在 Horovod 的 driver 进程里。driver 进程拿到所有 worker 进程节点的地址和 GPU 卡数信息后，会将其写入RendezvousServer 的 KVStore 中，然后 worker 就可以调用 gloo 来访问 RendezvousServer 构造通信环。
- Driver 会在 worker 节点上启动/重启 worker 进程。
- Driver 会监控系统整体状态。

worker 负责训练和模型迭代。

- 每个 worker 节点会向 RendezvousServer 发起请求来得到自己的邻居节点信息，从而构造通信环。
- 在这个通信环之中，每个 worker 节点有一个左邻居和一个右邻居，在通信过程中，每个 worker 只会向它的右邻居发送数据，只会从左邻居接受数据。

具体组网机制如下：





我们下面详细分析下各个部分。

1.2 容错机制

Horovod 的容错机制是基于 gloo 来实现的，对于错误来说，这可以被认为是一个被动操作。

Gloo 本身是不支持容错的。当众多worker之间对张量进行聚合操作时候，如果某一个worker失败，则gloo不会处理异常，而是抛出异常并且退出，这样所有worker都会报异常退出。

为了不让某一个 worker 的失败导致整体训练退出，Horovod 需要做两方面工作：

- 不让异常影响现有作业
  。Horovod 必须捕获 gloo 抛出的异常，于是就构建了一个python处理异常机制。
    - Worker 在捕获异常之后会将异常传递给对应的 Python API 处理，API 通过判断异常类型决定是否继续训练。
    - 如果异常信息中包括 “HorovodAllreduce”、“HorovodAllgather” 或者 “HorovodBroadcast” 等关键字，说明这可能是某个worker死掉导致的通信失败，这种异常被Horovod认为是可以恢复的。
- 放弃失败的worker，使用剩余可用worker继续训练
    - 其他存活的 worker 停止当前的训练，记录当前模型迭代的步数。
    - 此时gloo的runtime已经出现问题，通信环已经破裂，无法在剩余的 worker 之间继续进行 AllReduce 操作。
    - 为了可以继续训练，Horovod Driver 会重新初始化 gloo，启动一个新的 rendezvous server，然后获取存活的 worker 的信息，利用这些worker组成新的通信环。
    - 当新的通信环构造成功后，rank 0 worker 会把自己的模型广播发给其他所有worker，这样大家就可以在一个基础上，接着上次停止的迭代开始训练。

1.4 监控机制

容错机制是被动操作，监控机制就是主动操作。

弹性就意味着分布式集群的状态会随时发生变化，而 Horovod 本身和分布式集群并没有关联，所以需要有一个外部途径来让 Horovod 随时掌握集群状态。

这个外部途径就是用户需要在 Horovod 启动命令中提供一个发现脚本 discovery_host。discovery_host 由用户编写，负责发现可用的 worker 节点拓扑信息。

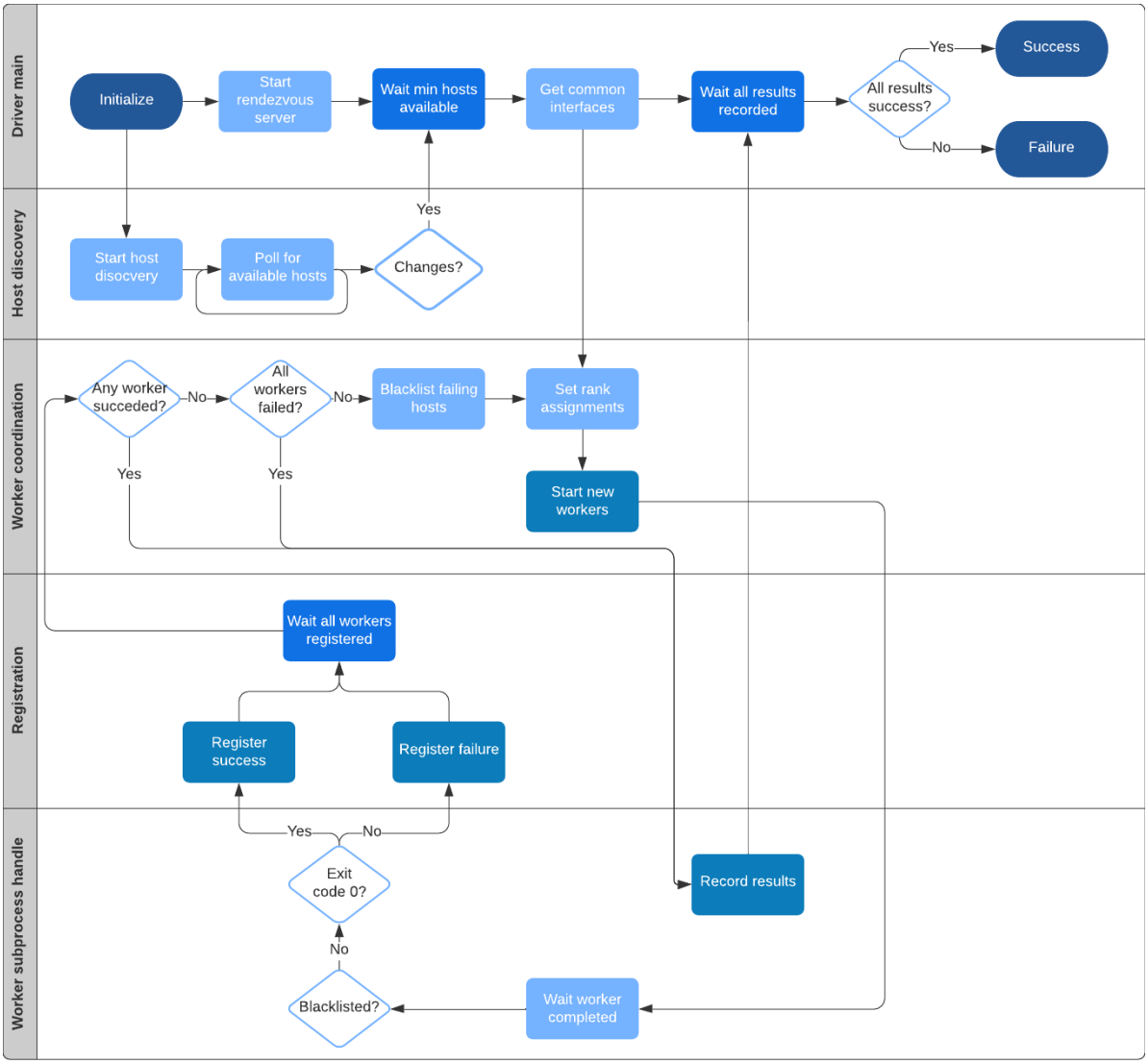
Driver在运行之后会定期调用这个 bash 脚本来对集群监控，当worker发生变化时，discover_host 脚本会返回最新的worker状态，Driver 根据 discover_host 的返回值得到 worker 节点信息：

- 如果Driver发现有worker失败，就捕获异常，根据存活的worker信息来更新 RendezvousServer KVStore 的节点信息，号召大家重新建立通信环进行训练。
- 如果Driver发现有新worker节点加入集群，根据目前所有worker信息来更新 RendezvousServer KVStore 的节点信息，号召大家重新建立通信环进行训练。现有worker 节点收到通知后，会暂停当前训练，记录目前迭代步数，调用 `shutdown` 和 `init` 重新构造通信环。Driver也会在新节点上启动worker，扩充进程数目。
- 当新的通信环构造成功之后，rank 0 worker 会把自己的模型广播发给其他所有worker，这样大家就可以在一个基础上，接着上次停止的迭代开始训练。

这样在训练过程中，当 worker 数量有变化时，训练依然继续进行。

1.5 官方架构图

官方的一个架构图如下，我们会在后续文章中逐步讲解图中部分：



0x02 示例代码

2.1 python代码

我们从官方文档中找出 TF v2 的示例代码看看，其关键之处是使用 @hvd.elastic.run 对 train 做了一个封装，并且传入了一个 TensorFlowKerasState。

```
import tensorflow as tf
import horovod.tensorflow as hvd

hvd.init()

gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
if gpus:
    tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')

dataset = ...
model = ...

optimizer = tf.optimizers.Adam(lr * hvd.size())

@tf.function
def train_one_batch(data, target, allreduce=True):
    with tf.GradientTape() as tape:
        probs = model(data, training=True)
        loss = tf.losses.categorical_crossentropy(target, probs)

    if allreduce:
        tape = hvd.DistributedGradientTape(tape)

    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

# Initialize model and optimizer state so we can synchronize across workers
data, target = get_random_batch()
train_one_batch(data, target, allreduce=False)

# 使用 @hvd.elastic.run 对 train 做了一个封装
@hvd.elastic.run
def train(state):
    for state.epoch in range(state.epoch, epochs):
        for state.batch in range(state.batch, batches_per_epoch):
            data, target = get_random_batch()
            train_one_batch(data, target)
            if state.batch % batches_per_commit == 0:
                state.commit()
        state.batch = 0

def on_state_reset():
    optimizer.lr.assign(lr * hvd.size())

# 这里是新修改处，传入了一个 TensorFlowKerasState
state = hvd.elastic.TensorFlowKerasState(model, optimizer, batch=0, epoch=0)
state.register_reset_callbacks([on_state_reset])
train(state)
```

2.2 脚本执行

弹性训练依然使用 **horovodrun** 这个命令行工具跑，和普通分布式训练不同的是，弹性训练不会在启动命令中明确指定节点列表，而是是使用一个 **发现机制** 来在运行时发现节点。通用的做法是在启动 Job 时候提供一个发现脚本:

```
horovodrun -np 18 --host-discovery-script discover_hosts.sh python train.py
```

此脚本用以实时反馈当前可用的 hosts 以及每个 hosts 上的 slots（下文使用 `discover_hosts.sh` 指代该脚本，但其无需命名为 `discover_hosts.sh`）。

discover_hosts.sh 脚本必须有可执行权限，在被执行时返回可用节点列表，一行一个节点信息，结构为: ，例如:

```
$ sh ./discover_hosts.sh     # 运行脚本，输出节点信息
host-1:4
host-2:4
host-3:4
```

如果这个发现脚本运行失败（没有可执行权限）或者运行时返回非0错误码，则训练进程会立刻失败，否则会一直重试直到超时（返回的slot列表不满足最小可运行数）。

弹性训练会一直等到所需最小slots数（-np）准备好之后，才会开始运行训练进程，用户可以通过 **--min-np** 和 **--max-np** 指定最小和最大的slots数，如:

```
horovodrun -np 8 --min-np 4 --max-np 12 --host-discovery-script
discover_hosts.sh python train.py
```

如果可用slots数小于 **--min-np** 指定的数量时（比如某些节点故障，任务被抢占等），任务会被暂停等待，直到更多的节点变为活跃，或者超时时间 HOROVOD_ELASTIC_TIMEOUT（默认设置为600秒）达到。另外，如果不指定 **--min-np** ，则最小slots数会被默认为 **-np** 所配置的数目。

需要 **--max-np** 的原因是为了限制进程数目（防止过度使用可用资源），另外在学习率和数据分区方面也可以作为参考点（在这些情况下需要有一个固定的参考配置）。同样，如果不指定此参数，也会默认为 **--np** 。

0x03 逻辑流程

3.1 逻辑流程

我们先解析下弹性训练的逻辑流程（为了实现弹性训练的能力，Horovod Elastic 对 Horovod 的架构和实现进行了一定的修改），最大的差别就是：弹性训练需要在增删worker时候可以跟踪和同步worker的状态，具体修改如下。

1. 聚合操作需要被定义在 `hvd.elastic.run` 函数之下
    。
    1. 将你的主训练进程代码（初始化之后的所有代码）用一个函数（我们暂时命名为 train_func）封装起来，然后使用装饰器 hvd.elastic.run 装饰这个函数。
    2. 对于这个装饰器修饰的 train_func 函数，它第一个参数，必须是 hvd.elastic.State 的实例。因为某些新加入的worker可能会处于某些不确定的状态之中，所以在运行这个被装饰函数 train_func 之前，这个状态对象需要在所有worker中进行同步，以此确保所有的worker都达到一致状态。

3. 因为同步函数会用到集合通信操作，并且添加worker后，活跃worker不会在此函数之前重置，所以不要在同步函数之前使用Horovod的集合操作（比如broadcast, allreduce, allgather）。

2. 每个 worker 都有自己的状态（state）

   。

   1. 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。
   2. 对于TensorFlow，Keras和PyTorch，已经提供默认的标准状态实现。然而，如果用户需要在某些场景广播特殊类型，可以重载定制 hvd.elastic.State 这个对象。
   3. 在运行`hvd.elastic.run`函数前，此状态对象将在所有workers中同步一次，用于保持一致性。

3. 周期性调用 state.commit() 来把状态（state）备份到内存

   。

   1. 定期备份非常有用。在某些worker发生意外错误时，定期备份可以避免因为状态被损坏而在重新训练时候无法恢复现场。比如，如果一个worker刚好在更新参数过程中突然出错，此时部分梯度更新完毕，部分梯度可能只更新到一半，这个状态是不可逆转而又无法继续。因此，当此状态发生时，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。
   2. 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。
   3. Elastic Horowod可以通过执行我们称之为“优雅地移除worker”操作来避免这些回滚。如果driver进程发现主机已可用或标记为删除，它将向所有workers推送一个通知。于是在下次调用`state.commit()`或更轻量级的`state.check_host_updates()`时，一个`HostsUpdatedInterrupt`异常将被抛出。此异常的处理方式与“HorovodInternalError”类似，只是参数状态不会还原到上次commit，而是从当前实时参数中恢复。
   4. 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。

4. 在 hvd.elastic.State 对象中注册一些回调函数，以便当worker成员发生变化时给予响应

   1. 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。

   2. 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。

5. worker 的增减会触发其他 worker 上的重置（reset）事件

   ，重置事件会激活以下几个操作（具体执行依据情况决定，不一定全部执行）：

   1. 判断该 worker 是否可以继续运行。
   2. 将失效的 worker host 加入到黑名单，下一次组网不会使用blacklist中的host。
   3. 在新的 hosts 上启动 worker 进程。
   4. 更新每个 worker 的 rank 信息。

6. **在重置之后，每个 worker 的状态会被同步**

3.2 入口点

从如下代码可知 hvd.elastic.run 就是 horovod/tensorflow/elastic.py 之中的 run 函数。

```
import horovod.tensorflow as hvd
@hvd.elastic.run
```

所以我们去这个文件中探寻。

```
def run(func):
    from tensorflow.python.framework.errors_impl import UnknownError

    def wrapper(state, *args, **kwargs):
        try:
            return func(state, *args, **kwargs)
        except UnknownError as e:
            if 'HorovodAllreduce' in e.message or \
                    'HorovodAllgather' in e.message or \
                    'HorovodBroadcast' in e.message:
                raise HorovodInternalError(e)
    return run_fn(wrapper, _reset)
```

3.3 主逻辑

run_fn 函数是关于用户代码的主要逻辑所在，位于 horovod/common/elastic.py。

其主要逻辑是：

- 初始化 notification_manager；
- 在 notification_manager 注册 state；
- 运行 func 函数，就是用户的训练代码 train；
- 在worker进程出现 HorvodInternalError 错误或者 HostsUpdateInterrupt 节点增删时，会捕获这两个错误，调用 reset 来进行容错处理；

```
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()
                    skip_sync = False
                except HostsUpdatedInterrupt as e:
                    skip_sync = e.skip_sync

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

3.4 出错处理

在出错状态下，当worker进程出现 **HorvodInternalError** （代表出现错误）或者 **HostsUpdateInterrupt** （代表有节点增删）时，Horovod 会执行如下流程:

1. 在 **hvd.elastic.run** 装饰器中捕获上述两个错误；
2. 如果抛出的是 **HorvodInternalError** 错误，则会从最后的一次 **commit** 状态中恢复；
3. 重新初始化 Horovod context，然后启动新的一轮的rendezvous，在rendezvous过程中，旧的worker会被优先被选举为新的rank-0，因为旧的worker具有上次训练中的最近状态；
4. 新的 rank-0 worker 会把状态同步到其它workers；
5. 继续训练；