(17) --- 弹性训练之容错

0x00 摘要

Horovod 是Uber于2017年发布的一个易于使用的高性能的分布式训练框架，在业界得到了广泛应用。

本系列将通过源码分析来带领大家了解 Horovod。本文是系列第十七篇，看看horovod 的容错机制。

我们依然用问题来引导学习。

问题是：

- 这些异常是 每个 worker 自动发出的吗？
- 是 worker 们一起抛出异常吗？
- 这些异常怎么通知给 Driver？

我们下面一一分析（为了可以独立成文，本文部分原理内容与前文相同）。

0x01总体思路

首先，我们需要注意的是：在某种程度上，**容错和弹性调度互为因果。**

- 容错的意思是，作业不受其中进程数量变化影响。
- 弹性调度时，作业里的进程数量会随集群 workload 情况增减，所以作业必须是容错的，才能和调度系统配合，实现弹性调度。

其次，在源码的文档之中，有如下注释，我们可以看到容错具体思路。

```
The reset process following a ``HorovodInternalError`` (failure) or
``HostsUpdatedInterrupt`` (add/remove request) is as follows:

1. Catch exception within the ``hvd.elastic.run`` decorator.
2. Restore last committed state if ``HorovodInternalError`` was raised.
3. Reinitialize Horovod context performing a new round of rendezvous.
4. Synchronize state among the workers by broadcasting from the new worker-0.
5. Resume training by executing the underlying training function.

During rendezvous, older workers will take priority in being assigned worker-0
status to ensure that the state that is broadcast is up to date.
```

大致翻译如下：

对于出错状态下，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：

- 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（commit）的状态；
- 重新初始化 Horovod context，然后driver 会根据当前正在运行的节点触发新的一轮的 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才具有最新的状态；
- 当新的通信域构造成功后，rank=0 的 worker 会将自身的模型（状态）广播给其他 worker；
- 接着上次停止的迭代步数开始训练，继续跑下训练函数(train)中的代码；

我们具体来看看如何处理。

0x02 抛出异常

2.1 示例代码

我们首先回顾下用示例代码。

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

hvd.init()

@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))

....

@hvd.elastic.run # 这里进行了包装，所以才能进行弹性训练
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

state = hvd.elastic.TensorFlowKerasState(model, optimizer, batch=0, epoch=0)
state.register_reset_callbacks([on_state_reset])
train(state)
```

最关键的就是用适配器 @hvd.elastic.run 包装了 train(state)，所以我们顺着来看。

2.2 HorovodInternalError

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
```

因此我们来到了 horovod/tensorflow/elastic.py。

func 就是用户训练函数，**当运行用户训练函数出错时候**，会根据捕获的异常信息来进行分析，如果是 ring allreduce 相关，就转为抛出异常 HorovodInternalError(e)。

```
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:
            # 判断是否是集合通信相关
            if 'HorovodAllreduce' in e.message or \
                    'HorovodAllgather' in e.message or \
                    'HorovodBroadcast' in e.message:
                raise HorovodInternalError(e)
    return run_fn(wrapper, _reset)
```

2.3 HostsUpdatedInterrupt

从前文我们知道：

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，worker 根据通知来进行处理。

具体如下：

- 驱动（后台发现）进程 获取 WorkerNotificationClient，然后调用 WorkerNotificationClient 来进行通知。就是利用 WorkerNotificationClient 发送 HostsUpdatedRequest。
- 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。
- WorkerNotificationService 会响应 HostsUpdatedRequest。调用 handle_hosts_updated 会逐一通知注册在WorkerNotificationManager 上的 listener（就是用户代码中的 State）。
- 每一个 worker 有自己对应的 State，都位于 `WorkerNotificationManager . _listeners`。

- 每个worker收到通知之后，调用 _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。
- 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）。
- state.check_host_updates() 会抛出 **HostsUpdateInterrupt** 异常。

具体代码如下：

在用户调用 commit 的时候，会调用 check_host_updates 检查更新。这里对用户代码是侵入了，用户使用到了框架的东西，虽然不知道 Driver，但是用到了框架的其他东西，比如 state。

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

检查更新如下。

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

    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:
        # 抛出异常
        raise HostsUpdatedInterrupt(all_update == HostUpdateResult.removed)
```

2.4 总结

因此我们可以回答文初的两个问题：

- 这些异常是 每个 worker 自动发出的吗？
    - 是的自动抛出的。
    - 当运行 用户训练函数出错时候，会根据捕获的异常信息来进行分析，如果是 ring allreduce 相关，就转为抛出异常 HorovodInternalError(e)。
    - 当如果发现 host 有变化，就会产生一个 HostsUpdatedInterrupt 异常。
- 是 worker 们一起抛出异常吗？
    - 是一起抛出。
    - 如果训练出错，则都会抛出异常

- 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**一起**抛出一个 HostsUpdateInterrupt 异常。

抛出异常的逻辑如下：

0x03 处理异常

3.1 总体逻辑

总体架构是 在 run_fn 之中。

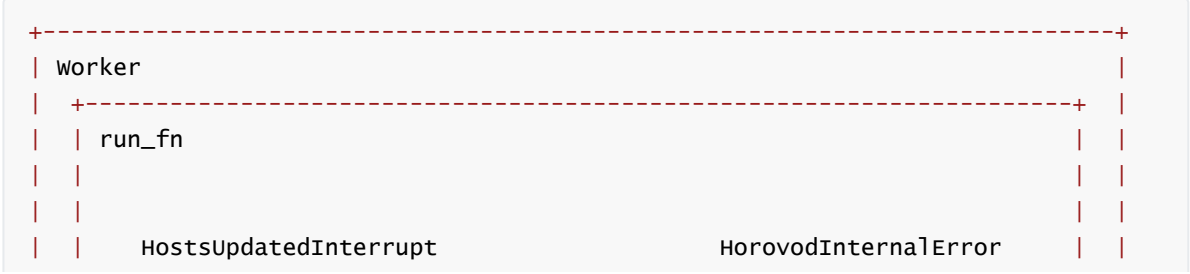
回忆一下 run_fn 是从哪里来调用的。原来是在 run 之中，就是运行 wrapper。而 wrapper 本身是对用户训练函数的包装。

```
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)
```

大概逻辑如图：



run_fn逻辑如下：

- 当 HorovodInternalError 产生，就会调用 state.restore() 来恢复；
- 当 HostsUpdatedInterrupt 被捕获，会设置 skip_sync；
- 调用 reset()，state.on_reset() 进行重置；
- 当下次循环，会根据 skip_sync 决定是否执行 state.sync()；

具体代码如下：

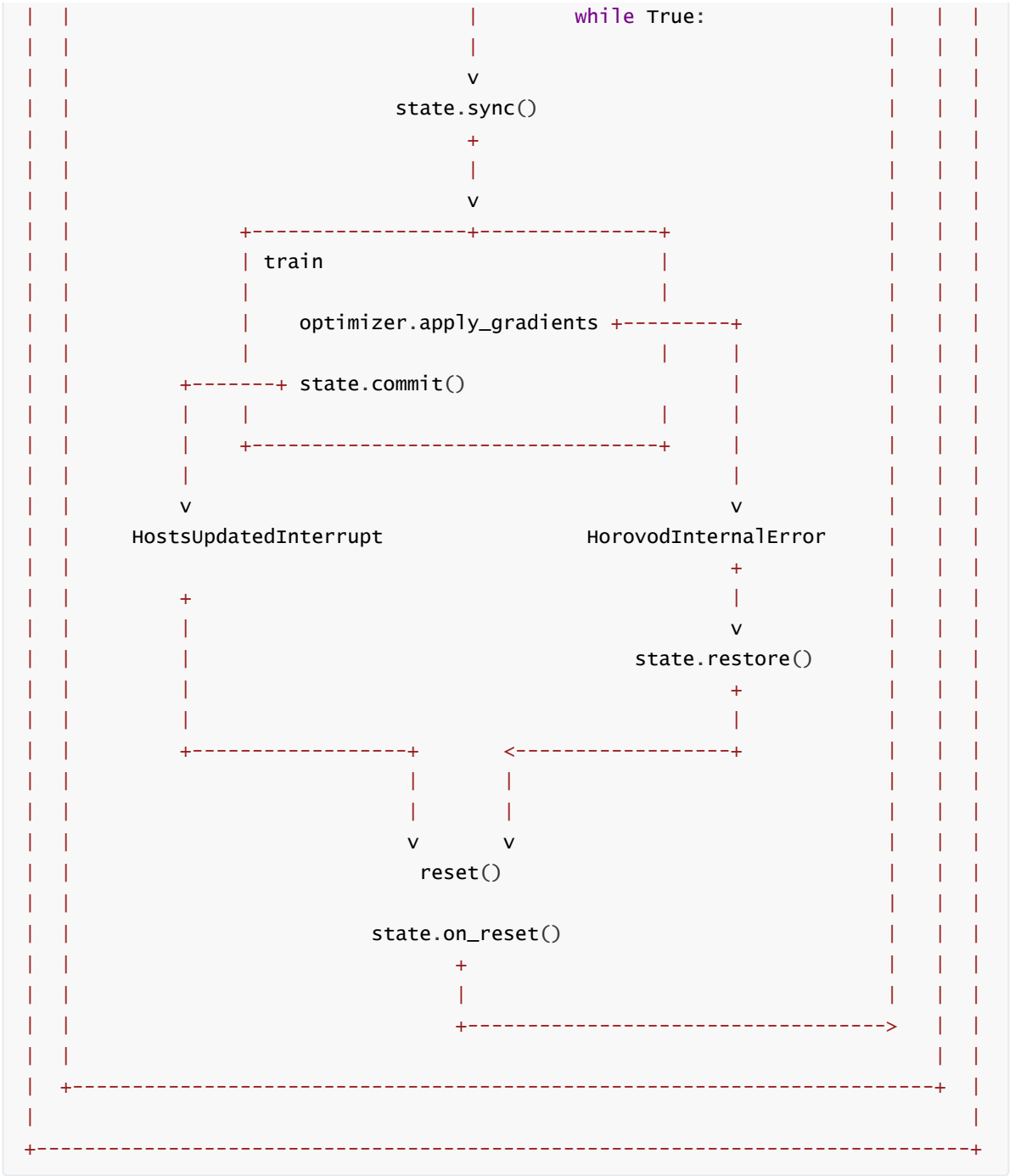
```
def run_fn(func, reset):
    @functools.wraps(func)
    def wrapper(state, *args, **kwargs):
        notification_manager.init()
        notification_manager.register_listener(state)
        skip_sync = False

        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
```

所以我们拓展逻辑如下：



3.2 恢复

state.restore() 会进行恢复。

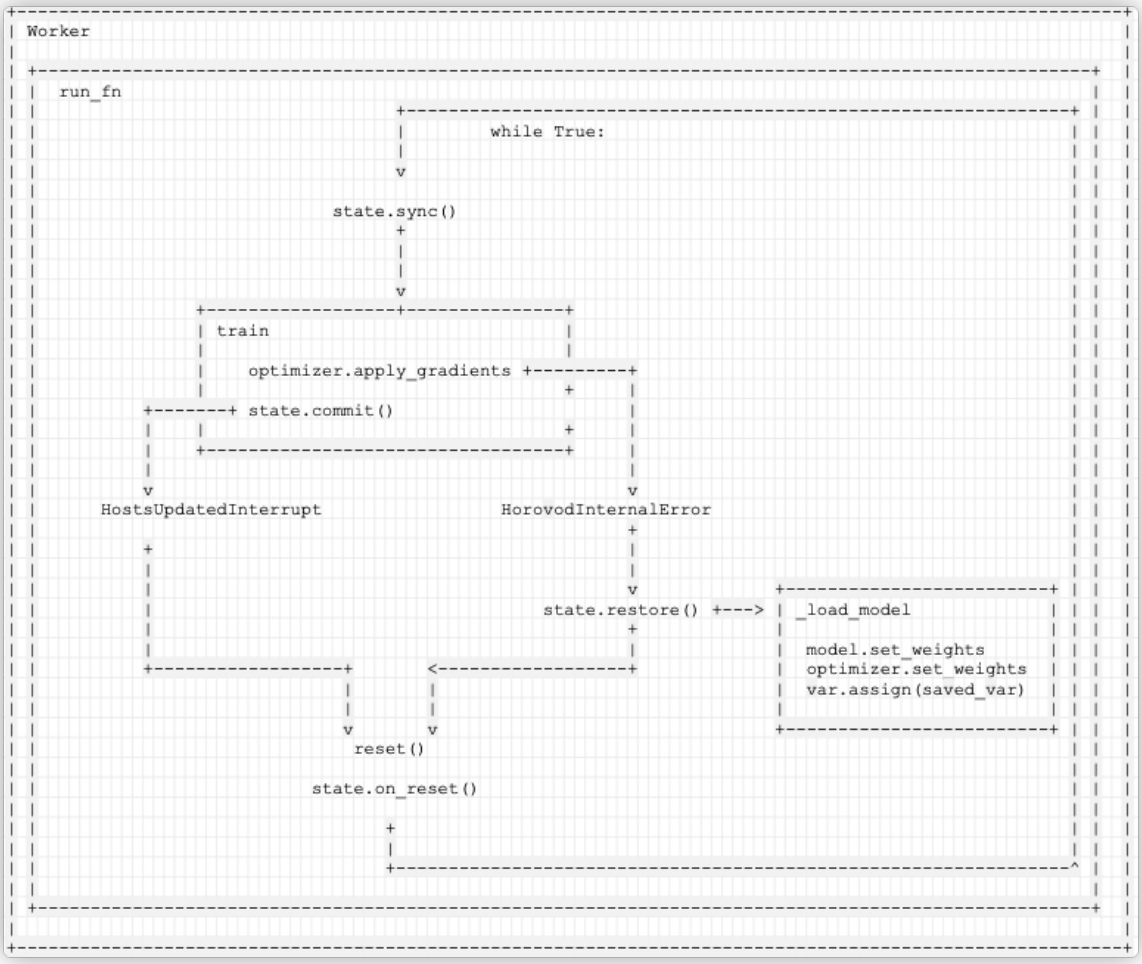
在 TensorFlowKerasState 之中，实现了 restore。

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

具体 restore 就是重新加载模型，具体加载就是利用 TensorFlowKerasState 的 model, optimizer 这两个成员变量。

```
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)
```

我们拓展如下：



3.3 重置

以下代码会进行重置操作。

```
reset()
state.on_reset()
```

3.3.1 reset

具体 reset 函数是：

```
def _reset():
    shutdown()
    init()
```

3.3.2 _HorovodBasics

具体使用了 _HorovodBasics 这里的函数。

```
_basics = _HorovodBasics(__file__, 'mpi_lib')

init = _basics.init
shutdown = _basics.shutdown
```

具体如下，就是重新建立 MPI 相关 context。

```
def init(self, comm=None):

    if comm is None:
        comm = []

    atexit.register(self.shutdown)

    if not isinstance(comm, list):
        mpi_built = self.MPI_LIB_CTYPES.horovod_mpi_built()

        from mpi4py import MPI
        if MPI._sizeof(MPI.Comm) == ctypes.sizeof(ctypes.c_int):
            MPI_Comm = ctypes.c_int
        else:
            MPI_Comm = ctypes.c_void_p
            self.MPI_LIB_CTYPES.horovod_init_comm.argtypes = [MPI_Comm]

        comm_obj = MPI_Comm.from_address(MPI._addressof(comm))
        self.MPI_LIB_CTYPES.horovod_init_comm(comm_obj)
    else:
        comm_size = len(comm)
        self.MPI_LIB_CTYPES.horovod_init(
            (ctypes.c_int * comm_size)(*comm), ctypes.c_int(comm_size))

def shutdown(self):
    self.MPI_LIB_CTYPES.horovod_shutdown()
```

3.3.3 on_reset

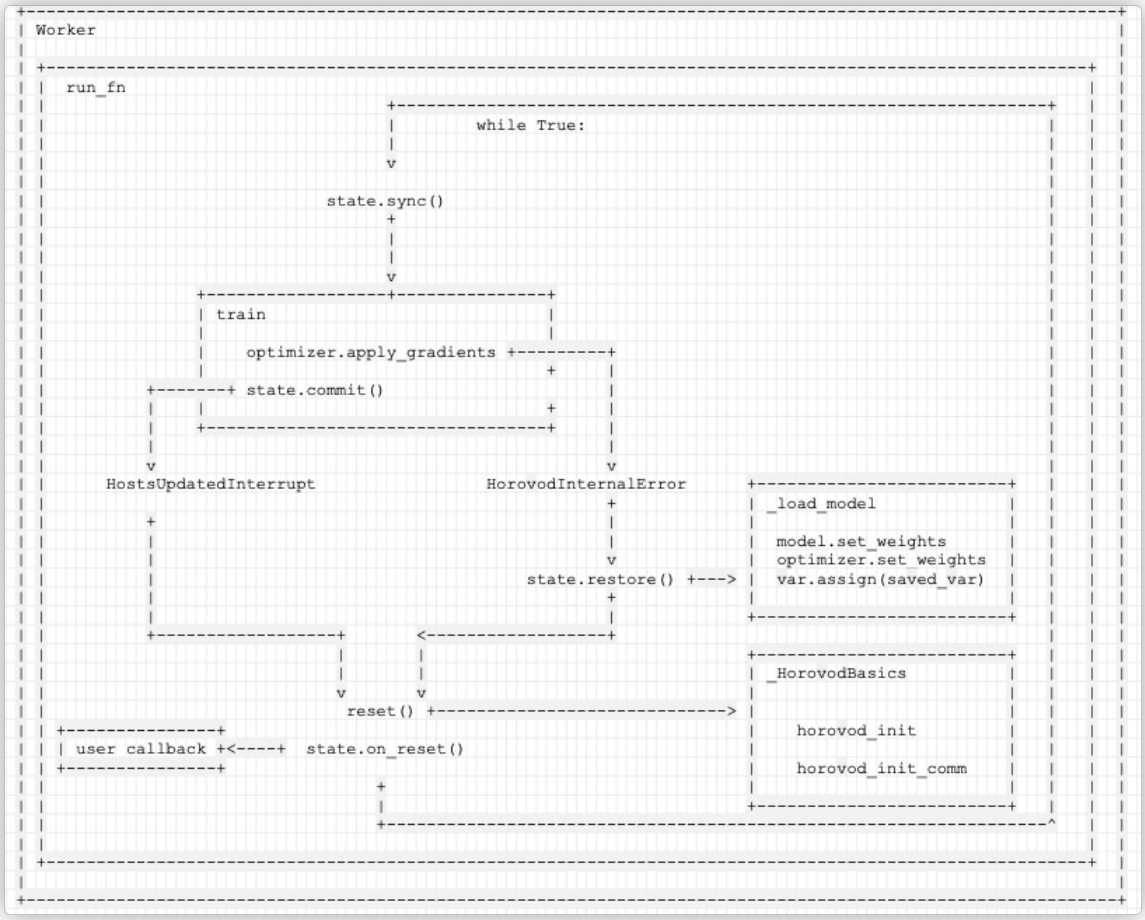
是执行用户设置的 reset callback。

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

比如用户设置如下callback：

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

此时逻辑如下：



3.3.4 sync

当重置时候，用户也会进行必要的同步，具体是广播变量 和 存模型 两步。

```
def sync(self):
    if self.session is not None:
        self.session.run(self._bcast_op)
    self._save_model()
    super(TensorFlowState, self).sync()
```

3.3.4.1 广播

广播函数在之前初始化时候有设置

```
self._bcast_op = broadcast_variables(self.variables, root_rank=0)
```

因此，就是 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。

3.3.4.2 存模型

存模型就是调用 _eval_fn 来把模型变量转存到内存之中。

```
def _save_model(self):
    self._values = [self._eval_fn(var) for var in self.variables]
```

_eval_fn 在 之前初始化时候有设置

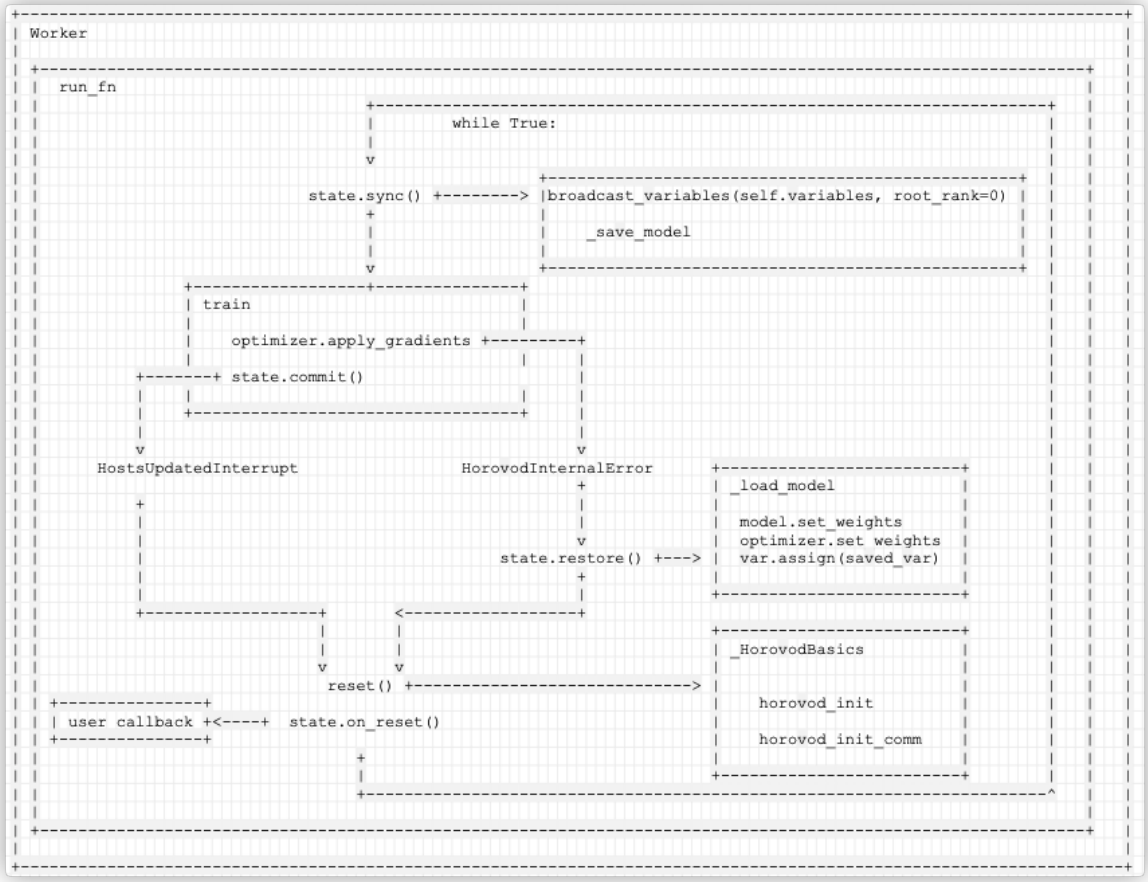
```
self._eval_fn = self._to_numpy if _executing_eagerly() else self._eval_var
```

具体函数是：

```
def _eval_var(self, var):
    return var.eval(self.session)

def _to_numpy(self, var):
    return var.numpy()
```

所以我们的逻辑拓展如下：



至此，弹性训练部分分析结束。下面二~三篇文章将为大家介绍K8S相关。