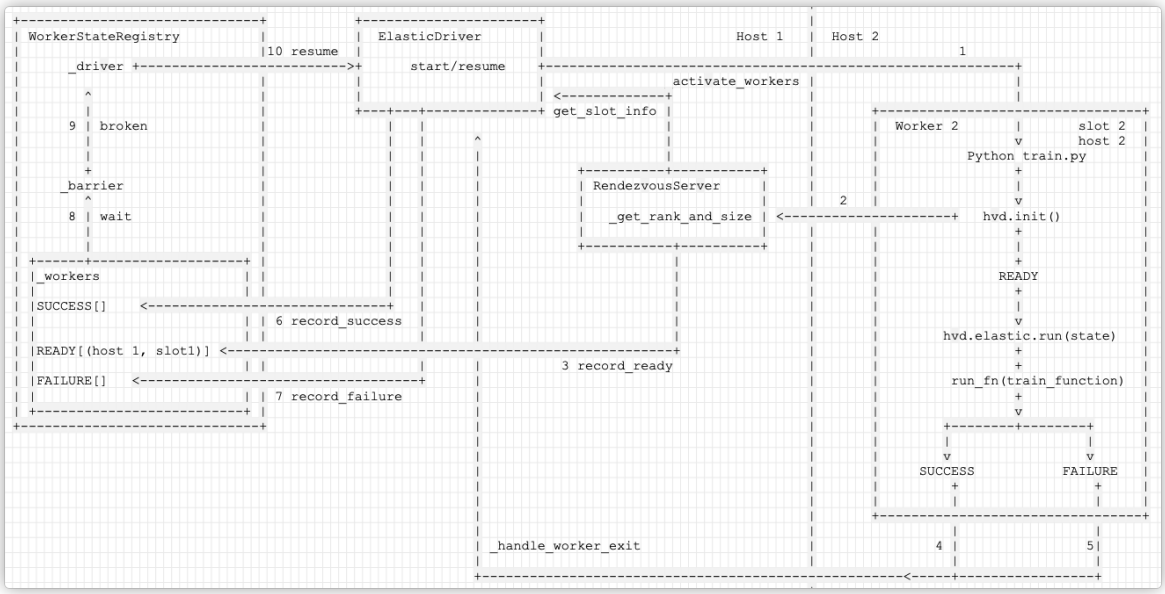
horovod (16) --- 弹性训练之Worker生命周期

目录

0x00 摘要

本文是第十六篇，看看 horovod **弹性训练**中 worker 的生命周期。

我们先给出一个逻辑图，大家先有一个粗略的了解，本图左侧是 Driver 部分，右侧是一个 Worker。



0x01 Worker 是什么

首先，我们要看看 worker 是什么。为了可以单独成文，本章节回忆了很多之前的知识，看过之前文章的同学可以跳过。

1.1 角色

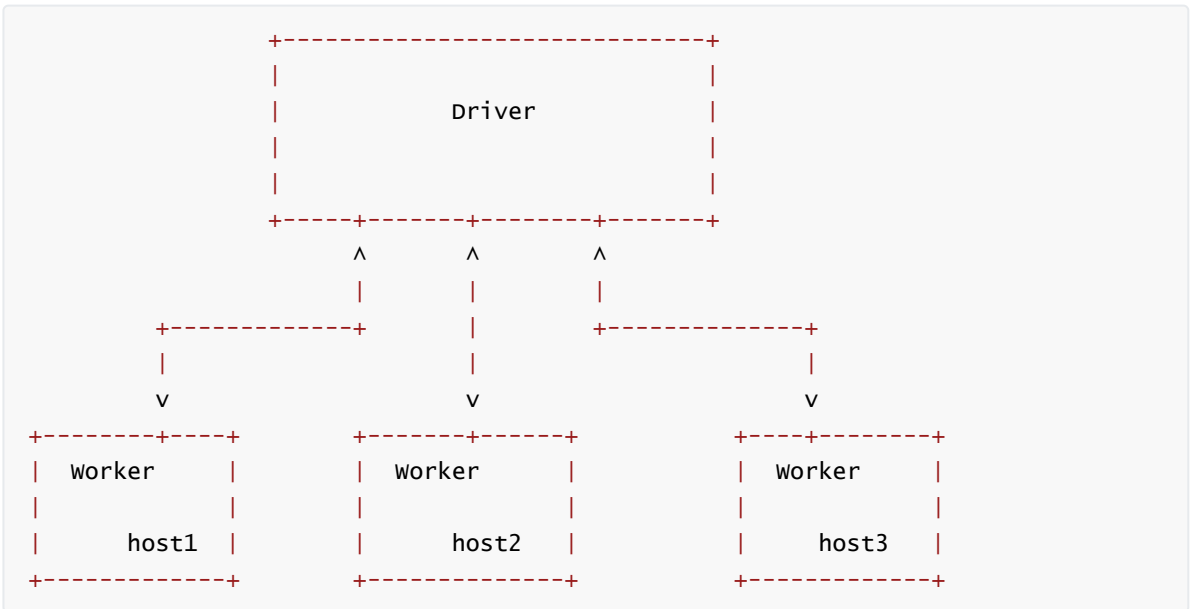
”训练“ 是通过计算梯度下降的方式利用数据来迭代地优化神经网络参数，最终输出网络模型的过程。

我们首先要看看弹性训练中的角色设定。

Horovod 的弹性训练包含两个角色，driver 进程和 worker 进程。driver 进程运行在 CPU 节点上，worker 进程可运行在 CPU 或者 GPU 节点上。在 Horovod 中，训练进程是平等的参与者，每个 worker 进程既负责梯度的分发，也负责具体的梯度计算。

这两个角色和 Spark 的 Driver -- Executor 依然很类似。Driver 进程就可以认为是 Spark 的 Driver，或者说是 master 节点。Worker 就类似于 Spark 的 Executor。

具体如图：



1.2 职责

角色的职责如下：

master（主节点）职责：

- 负责实时探活 worker（工作节点）是否有变化，掉线情况；
- 负责实时监控 host 是否有变化；
- 负责分配任务到存活的worker（工作节点）；
- 在有进程失败导致 AllReduce 调用失败的 情况下，master 通过 blacklist 机制 组织剩下的活着的进程构造一个新的环。
- 如果有新 host 加入，则生成新的 worker，新 worker 和 旧 worker 一起构造成一个新的环。

worker（工作节点）职责：

- 负责汇报（其实是被动的，没有主动机制）当前worker（工作节点）的状态（就是训练完成情况）；
- 负责在该worker（工作节点）负责的数据上执行训练。

1.3 组网机制

Horovod 在单机的多个 GPU 上采用 NCCL 来通信，在多机之间通过 ring-based AllReduce 算法进行通信。

Horovod 的弹性训练是指多机的弹性训练。在多机的 ring-based 通信中的每个 worker 节点有一个左邻和一个右邻，每个 worker 只会向它的右邻居发送数据，并从左邻居接受数据。

1.3.1 通信环

Driver 进程用于帮助 worker 调用 gloo 构造 AllReduce 通信环。

当 Horovod 在调用 Gloo 来构造通信域时，Horovod 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 地址 和 给其在逻辑通信环分配的序号 rank 等信息。

构建过程如下：

- Driver 进程创建带有 KVStore 的 RendezvousServer，即这个 RendezvousServer 运行在 Horovod 的 driver 进程里。
- Driver 进程拿到所有 worker 进程节点的 IP 地址和 GPU 卡数信息后，会将其写入 RendezvousServer 的 KVStore 中。
- 每个 worker 节点会通过调用 gloo 从而 请求 RendezvousServer 获取自己的邻居节点信息（ip，port...），从而构造通信域。

1.3.2 弹性构建

当有 worker 失败或者新的 worker 加入训练时，每个 worker 会停止当前的训练，记录当前模型迭代的步数，并尝试重新初始化 AllReduce 的通信域。

1.3.2.1 Driver 监控

因为 driver 进程一直在监控 worker 的状态 和 host 节点情况，所以

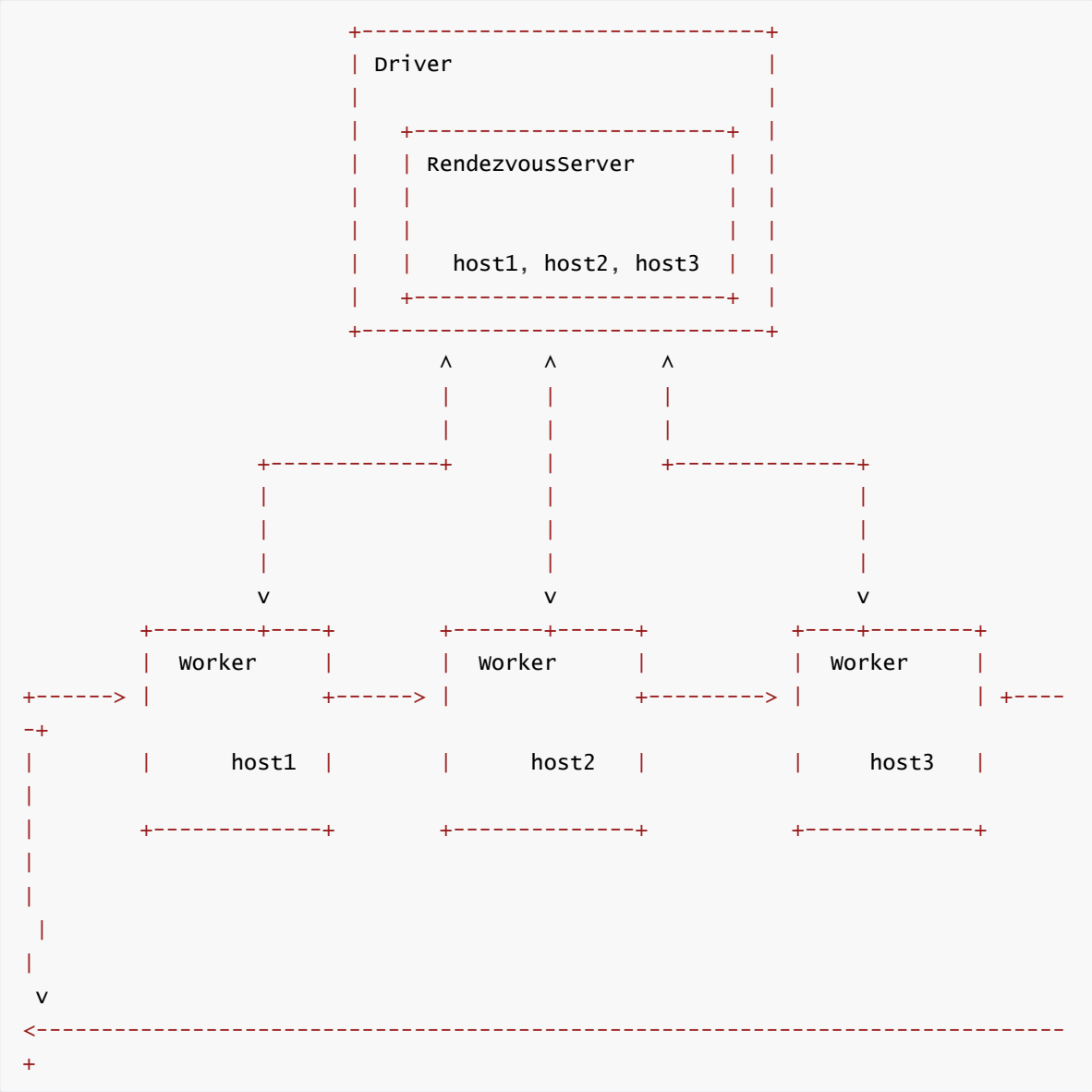
- 当 host 变化时候，当驱动进程通过节点发现脚本发现一个节点被标记为新增或者移除时，它将发送一个通知到 所有workers，在下一次 state.commit() 或者更轻量的 state.check_host_updates() 被调用时，会抛出一个 HostsUpdateInterrupt 异常。
- 当有 worker 失败时，driver 会重新获取存活的 worker 的host，

1.3.2.2 Driver 重新构建

为了不让其他 worker 进程退出，Horovod 会捕获 gloo 抛出的异常，并将异常传递给封装的 Python API。进而 driver 会重新配置 RendezvousServer，从而让 worker 节点能重新构造通信域。所以 Horovod 是可容错的。

如果有 host 变化，worker 进程可以通过这个 rendezvous 来构造新的通信域。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，然后接着上次停止的迭代步数开始训练。

组网机制如下：



所以，本文就看看 Worker 的一个整体生命流程。

0x02 总体生命流程

在 Driver 的 launch_gloo_elastic 之中，如下代码负责启动 worker。

- command 就是传入的可执行命令，比如 `python train.py`。经过 get_run_command 之后就得到了 env python train.py 之类的样子，就是加上环境变量，可以运行了。
- exec_command 类似如下： `exec_command = _exec_command_fn(settings)`，就是基于各种配置来生成可以执行命令环境。

```
run_command = get_run_command(command, server_ip, nics, global_rendezv_port,
elastic=True)

create_worker = _create_elastic_worker_fn(exec_command, run_command, env, event)

driver.start(settings.num_proc, create_worker)

res = driver.get_results()
```

可以清晰看出来三个详细的过程（因为 训练过程是在 worker 内部，所以 driver 分析没有深入此部分）：

- _create_elastic_worker_fn 是配置过程；
- driver.start 是启动过程；
- driver.get_results 就是得到 & 注册 运行结果；

我们接下来就按照这三个过程详细分析一下。

0x03 配置过程

配置过程 是由 _create_elastic_worker_fn 完成，就是提供一个在某个环境下运行某个命令的能力。

_create_elastic_worker_fn 分为两部分：

- _slot_info_to_command_fn 会建立 slot_info_to_command，套路和之前文章中类似，就是把各种 horovod 环境变量和运行命令 run_command 糅合起来，得到一个可以在“某个 host and slot”之上运行的命令文本；
- 返回 create_worker。
    - create_worker 是利用 exec_command 和 命令文本 构建的函数。
    - exec_command 我们在之前介绍过，就是提供了一种运行命令的能力，或者说是运行环境；
    - 所以 create_worker 就是提供一个在某个环境下运行某个命令的能力；

```
# 得到一个 可以在“某个 host and slot” 之上运行的命令文本
def _slot_info_to_command_fn(run_command, env):
     def slot_info_to_command(slot_info):
        """
        Given a slot_info, creates a command used by gloo to launch a single
job.
        :param slot_info: host and slot to execute the run command on
        """
        env_vars = create_slot_env_vars(slot_info)
        horovod_rendez_env = " ".join(
            [f"{k}={str(v)}" for k, v in env_vars.items()])

        return '{horovod_env} {env} {run_command}' .format(
            horovod_env=horovod_rendez_env,
            env=' '.join(['%s=%s' % (key, quote(value)) for key, value in
env.items()
                          if env_util.is_exportable(key)]),
            run_command=run_command)

    return slot_info_to_command

def _create_elastic_worker_fn(exec_command, run_command, env, event):
    get_command_with_env = _slot_info_to_command_fn(run_command, env)
```

```
    # 提供一个在某个环境下运行某个命令的能力
    def create_worker(slot_info, events):
        command = get_command_with_env(slot_info)
        events = [event] + (events or [])
        return exec_command(command, slot_info, events)
    return create_worker
```

所以，最终得到的 create_worker 是：

```
                command (python train.py)
                           +
                           |
                           |
          get_run_command  |
                           |
                           v
                run_command(env python train.py) #得到命令类似 env python train.py
                           +
                           |
                           |
 _slot_info_to_command_fn  |
                           |
                           v
              {horovod_env} {env} {run_command} #得到命令类似 horovod_env env python
train.py
                           +
                           |
                           |
              exec_command |
                           |
                           |
                           v
create_worker = exec_command({horovod_env} {env} {run_command})#得到在某个环境下运行
某个命令的能力
```

这样，create_worker 就直接可以运行训练代码了。

0x04 启动过程

`create_worker = _create_elastic_worker_fn` 提供了一个在某个环境下运行某个命令的能力，因为 `create_worker` 方法内部已经包括了执行命令和执行环境，就是说，只要运行`create_worker`，就可以自动训练。下面我们就利用这个能力来启动 worker。

启动过程基本都是在 ElasticDriver 类 的 start 方法中完成。

4.1 总体逻辑

以下逻辑都运行在 ElasticDriver 之中。

- 首先，会把 上面生成的 create_worker 赋值给 self._create_worker_fn。
- 其次，会调用 _activate_workers 启动多个 worker，其中包括：
  - 先使用 wait_for_available_slots 等待 min_np 数目的可用的 hosts。之前分析过此函数，就是 无限循环等待，如果 `avail_slots >= min_np and avail_hosts >= min_hosts` 才会返回。
  - 使用 _update_host_assignments 来得到 slots；

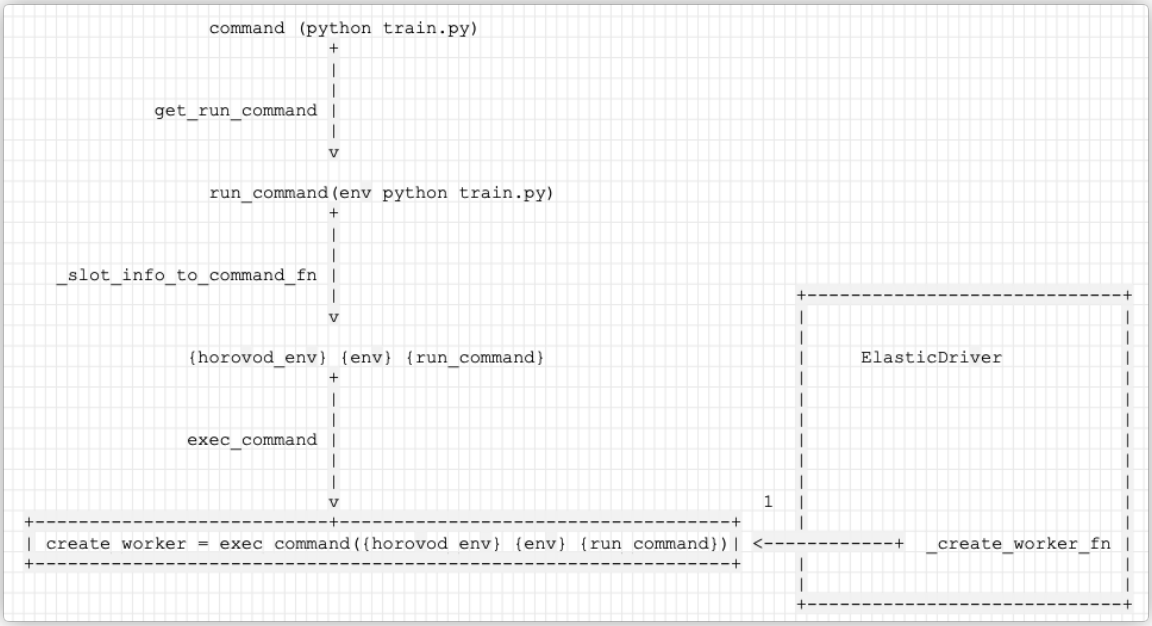
- 使用 _start_worker_processes 来启动多个 worker；

```
def _activate_workers(self, min_np):
    current_hosts = self.wait_for_available_slots(min_np)
    pending_slots = self._update_host_assignments(current_hosts)
    self._worker_registry.reset(self.world_size())
    self._start_worker_processes(pending_slots)

def start(self, np, create_worker_fn):
    self._create_worker_fn = create_worker_fn
    self._activate_workers(np)
```

下面逐一看看。

4.2 赋值

第一步就是上面生成的 create_worker 赋值给 self._create_worker_fn。



4.3 获取 host 信息

接下来要使用 _update_host_assignments 来得到 slots，具体分为两步：

首先构建 host 和 rank 之间的分配状况。

其次

4.3.1 更新 host 和 rank

_update_host_assignments 函数中会根据 最新的 host 信息，重新构建 rendezvous，比如：

`self._rendezvous.init(host_assignments_list)`。

具体逻辑是：

- 获取 活跃的slot： `active_slots`；
- 获取 host 分配情况；
- 确保每个 worker 有前驱者，就是可以传递状态，构成环；
- 调用 self._rendezvous.init 重新构造 rendezvous；
- 分配 rank 和 slot 的关系；

- 返回 pending_slots，就是分配的slot之中，不在 活跃slot列表 `active_slots` 中的。不活跃的就是接下来可以启动 新 worker 的。

```
def _update_host_assignments(self, current_hosts):
    # Determine the slots that are already filled so we do not respawn these
processes
    # 获取 活跃的slot
    active_slots = set([(host, slot_info.local_rank)
                        for host, slots in self._host_assignments.items()
                        for slot_info in slots])

    # Adjust the host assignments to account for added / removed hosts
    host_assignments, host_assignments_list =
self._get_host_assignments(current_hosts)

    if len(self._host_assignments) > 0:
        # Ensure that at least one previously active host is still assigned,
otherwise there is no
        # way to sync the state to the new workers
        prev_hosts = self._host_assignments.keys()
        next_hosts = host_assignments.keys()
        if not prev_hosts & next_hosts:
            raise RuntimeError('No hosts from previous set remaining, unable to
broadcast state.')

    self._host_assignments = host_assignments
    self._world_size = len(host_assignments_list)

    self._rendezvous.init(host_assignments_list) # 重新构造 rendezvous

    # Rank assignments map from world rank to slot info
    rank_assignments = {}
    for slot_info in host_assignments_list:
        rank_assignments[slot_info.rank] = slot_info
    self._rank_assignments = rank_assignments

    # Get the newly assigned slots that need to be started
    pending_slots = [slot_info
                     for host, slots in self._host_assignments.items()
                     for slot_info in slots
                     if (host, slot_info.local_rank) not in active_slots]
    return pending_slots
```

4.3.2 获取 host 和 rank

其中，host信息是通过 _get_host_assignments 来完成

```
def _get_host_assignments(self, current_hosts):
    # Adjust the host assignments to account for added / removed hosts
    host_list = [hosts.HostInfo(host, current_hosts.get_slots(host))
                 for host in current_hosts.host_assignment_order]
    host_assignments_list = hosts.get_host_assignments(host_list, self._min_np,
self._max_np)
    host_assignments = defaultdict(list)
    for slot_info in host_assignments_list:
        host_assignments[slot_info.hostname].append(slot_info)
    return host_assignments, host_assignments_list
```

_get_host_assignments 调用get_host_assignments具体完成业务。

get_host_assignments 会依据 host 和 process capacities (slots) 来给 Horovod 之中的进程分配，即给出一个 horovod rank 和 slot 的对应关系。设置了几个 np，就有几个 slot。

给出的分配方案类似如下，这样就知道了哪个rank对应于哪个host上的哪个slot：

```
[
  SlotInfo(hostname='h1', rank=0, local_rank=0, cross_rank=0, size=2,
local_size=2, coress_size=1),
    SlotInfo(hostname='h2', rank=1, local_rank=0, cross_rank=0, size=2,
local_size=2, coress_size=1),
]
```

代码如下：

```
def get_host_assignments(hosts, min_np, max_np=None):
    """Assign hosts with process capacities (slots) to ranks in the Horovod
process.
    This function will try to allocate as many as possible processes on the same
host to leverage local network.

    :param hosts: list of HostInfo objects describing host and slot capacity
    :type hosts: list[HostInfo]
    :param min_np: minimum number of processes to be allocated
    :param max_np: (optional) maximum number of processes to be allocated
    :return: a list of the allocation of process on hosts in a `SlotInfo`
object.
    :rtype: list[SlotInfo]
    """
    host_ranks = []
    cross_ranks = collections.defaultdict(dict)
    rank = 0
    # 依据 hosts 信息构建 rank, local rank, cross rank(hierarchical allreduce所需要)
    for host_info in hosts:
        ranks = []
        for local_rank in range(host_info.slots):
            if rank == max_np:
                break

            ranks.append(rank)
            rank += 1

            cross_ranks_at_local = cross_ranks[local_rank]
            cross_ranks_at_local[host_info.hostname] = len(cross_ranks_at_local)
```

```
        host_ranks.append((host_info, ranks))

    world_size = rank

    # 给出一个 horovod rank 和 slot 的对应关系。返回一个alloc_list，每个SlotInfo包括各种
rank信息
    alloc_list = []
    for host_info, ranks in host_ranks:
        local_size = len(ranks)
        for local_rank, rank in enumerate(ranks):
            cross_ranks_at_local = cross_ranks[local_rank]
            cross_rank = cross_ranks_at_local[host_info.hostname]
            cross_size = len(cross_ranks_at_local)

            alloc_list.append(
                SlotInfo(
                    hostname=host_info.hostname,
                    rank=rank,
                    local_rank=local_rank,
                    cross_rank=cross_rank,
                    size=world_size,
                    local_size=local_size,
                    cross_size=cross_size))

    return alloc_list
```

4.3.3 拓展逻辑

目前拓展逻辑如下，经过 4.2，4.3 两步之后，ElasticDriver 中的一些变量被赋值了，我们简化了上图中的 exec_command 如下（第一步在上图中）：

```
+-----------------------------------+------------+
| exec_command({horovod_env} {env} {run_command})|
+-------------------------------------------------+
```

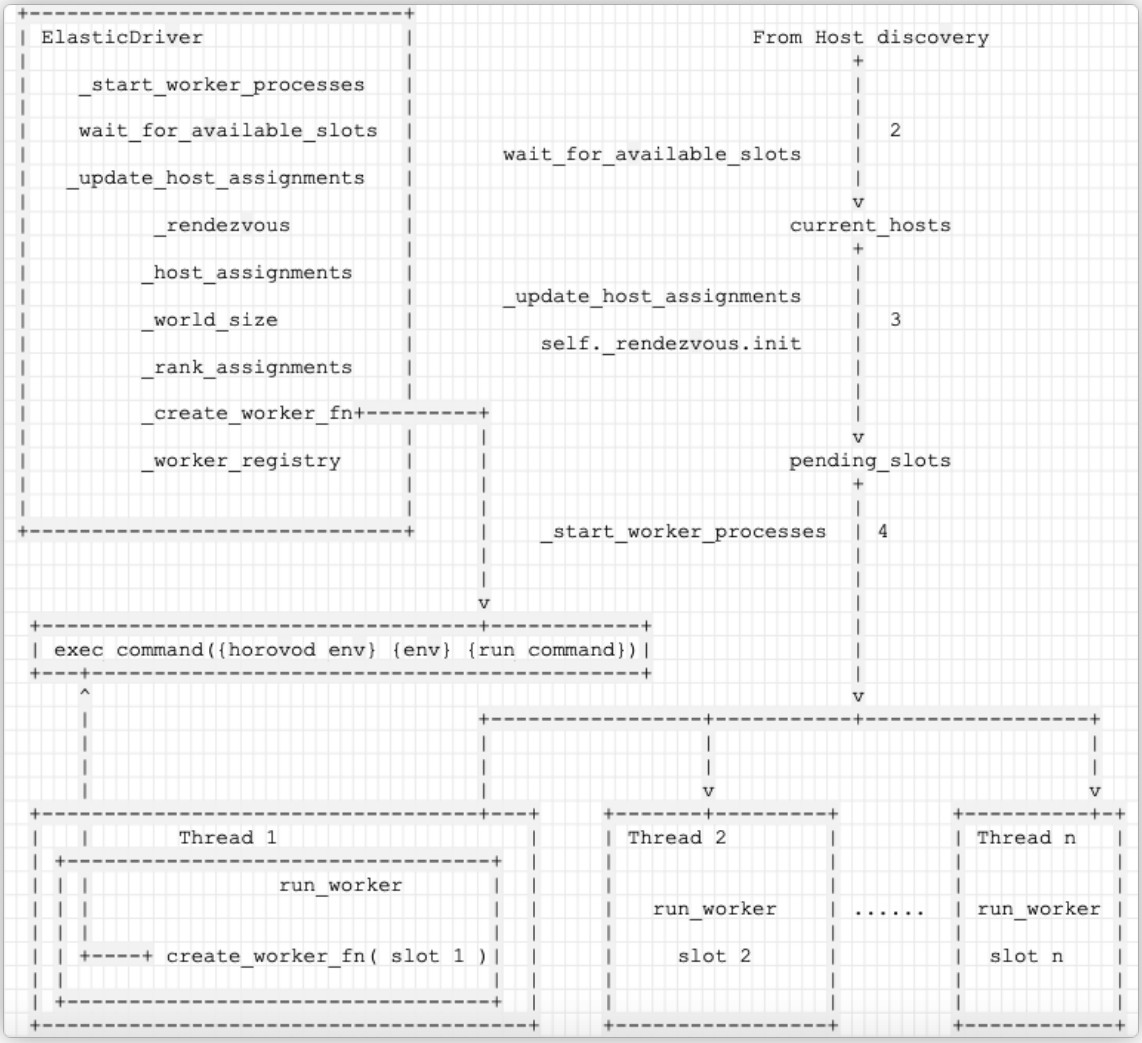
4.4 启动

_start_worker_processes 完成了启动过程，逻辑递增如下。

- create_worker_fn 就是使用 之前赋值的 create_worker = exec_command({horovod_env} {env} {run_command}) ；
- run_worker() 中 执行 create_worker_fn(slot_info, [shutdown_event, host_event]) 就是 运行训练代码；
- threading.Thread(target=run_worker) 就是在一个 thread 之中运行训练代码；
- _start_worker_processes 就是在多个 thread 之中运行多份训练代码；

```
def _start_worker_processes(self, pending_slots):
    for slot_info in pending_slots:
        self._start_worker_process(slot_info)

def _start_worker_process(self, slot_info):
    create_worker_fn = self._create_worker_fn
    shutdown_event = self._shutdown
    host_event = self._host_manager.get_host_event(slot_info.hostname)

    def run_worker():
        res = create_worker_fn(slot_info, [shutdown_event, host_event])
        exit_code, timestamp = res
        self._handle_worker_exit(slot_info, exit_code, timestamp)

    thread = threading.Thread(target=run_worker)
    thread.daemon = True
    thread.start()
    self._results.expect(thread)
```

启动之后，逻辑如下，可以看到，经过 4 步之后，启动了 count（slot_info）这么多的 Thread，每个 Thread 之中，有一个 _create_worker_fn 运行在一个 Slot 之上：



0x05 运行过程

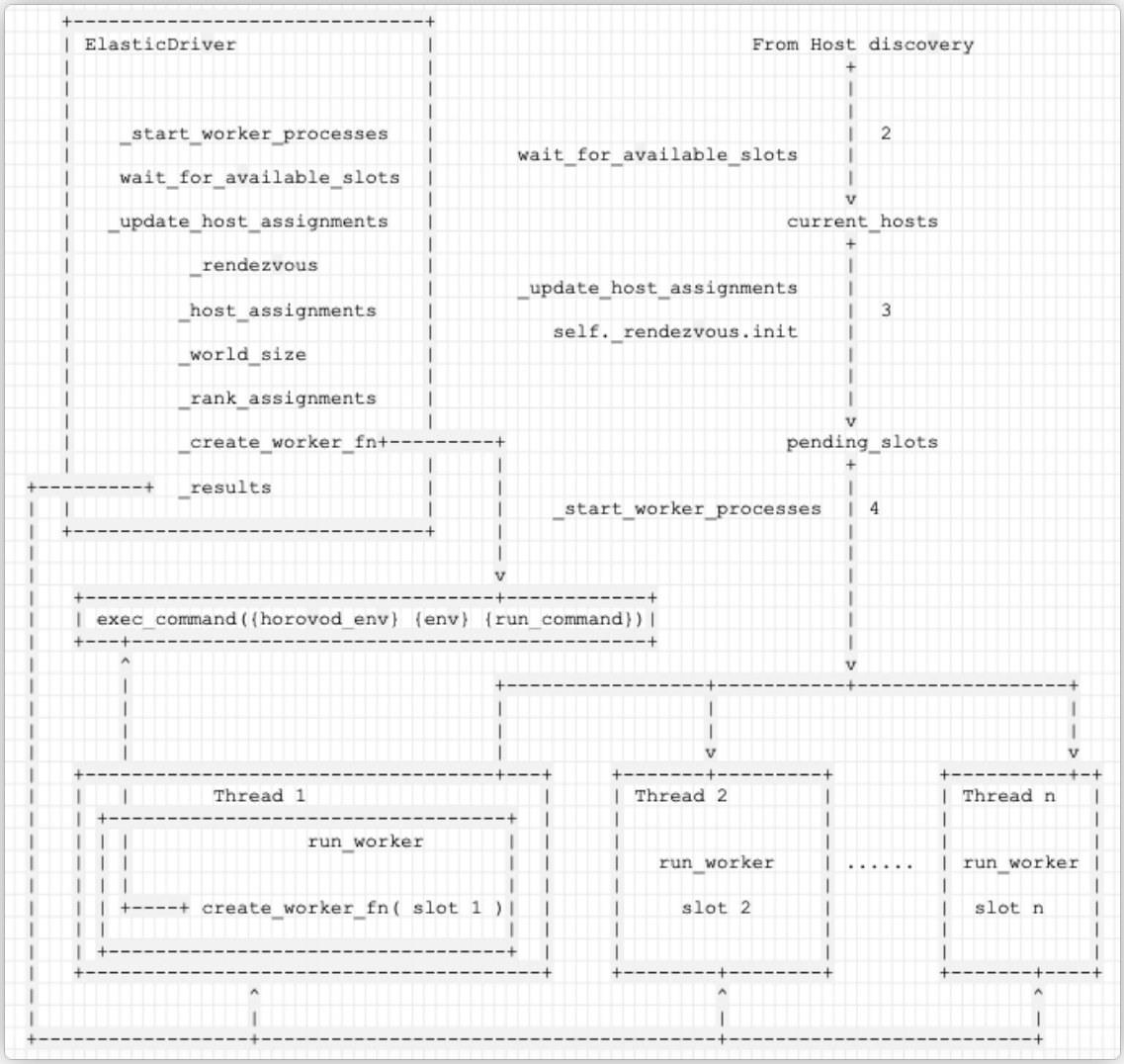
运行过程其实在上面已经提到了，就是在 Thread 之中运行 `exec_command({horovod_env} {env} {run_command})`。这就是调用用户代码进行训练。

```
thread = threading.Thread(target=run_worker)
thread.daemon = True
thread.start()
self._results.expect(thread)
```

这里要说明的是 self._results = ResultsRecorder() 。具体在其中记录每一个运行的 Thread。

```
class ResultsRecorder(object):
    def __init__(self):
        self._error_message = None
        self._worker_results = {}
        self._worker_threads = queue.Queue()

    def expect(self, worker_thread):
        self._worker_threads.put(worker_thread)
```

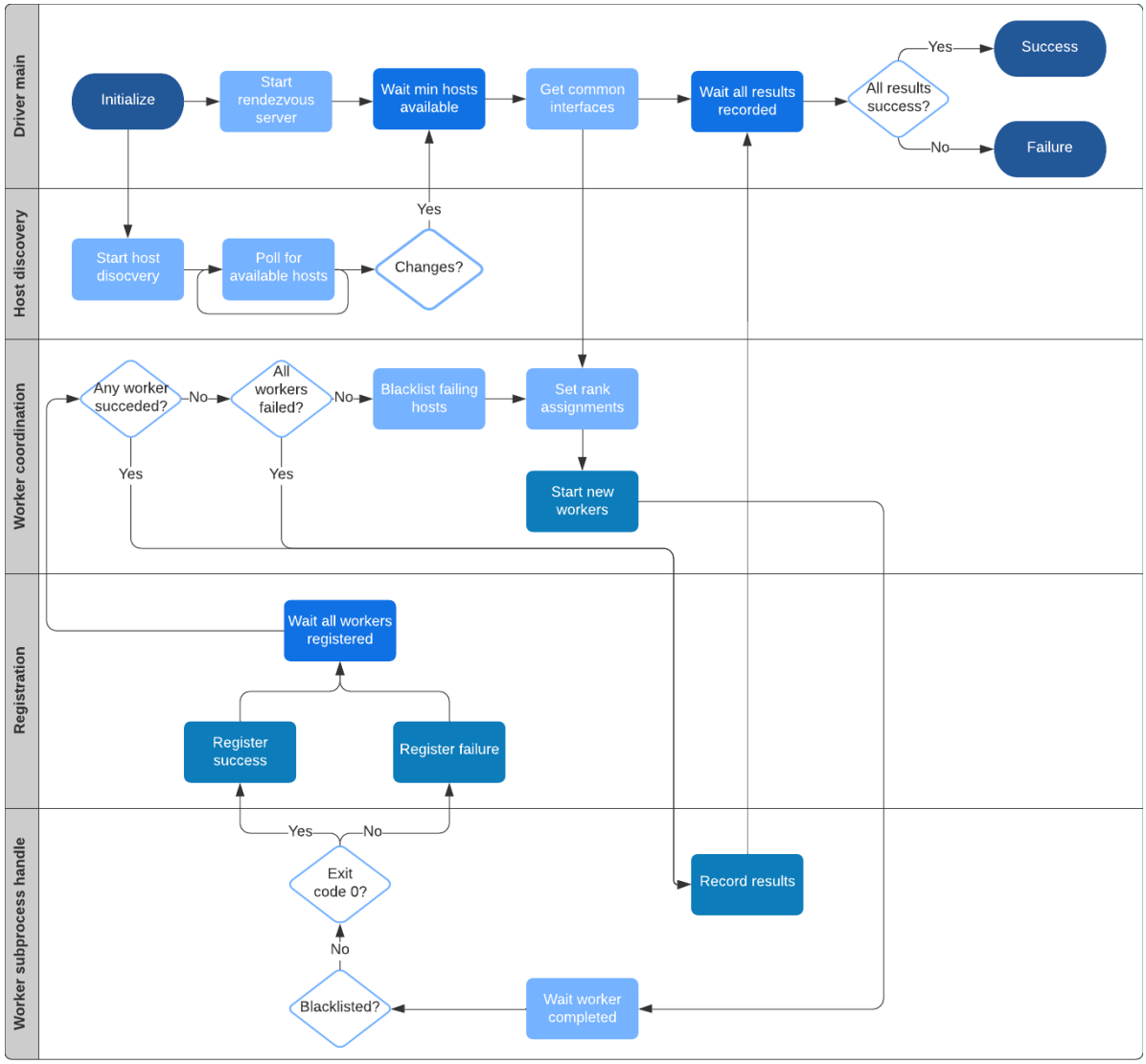
于是我们的逻辑变成如下，self._results 里面记录了所有的 Threads：



0x06 注册，结果 & 协调

本节主要对应架构图中的下面三部分。

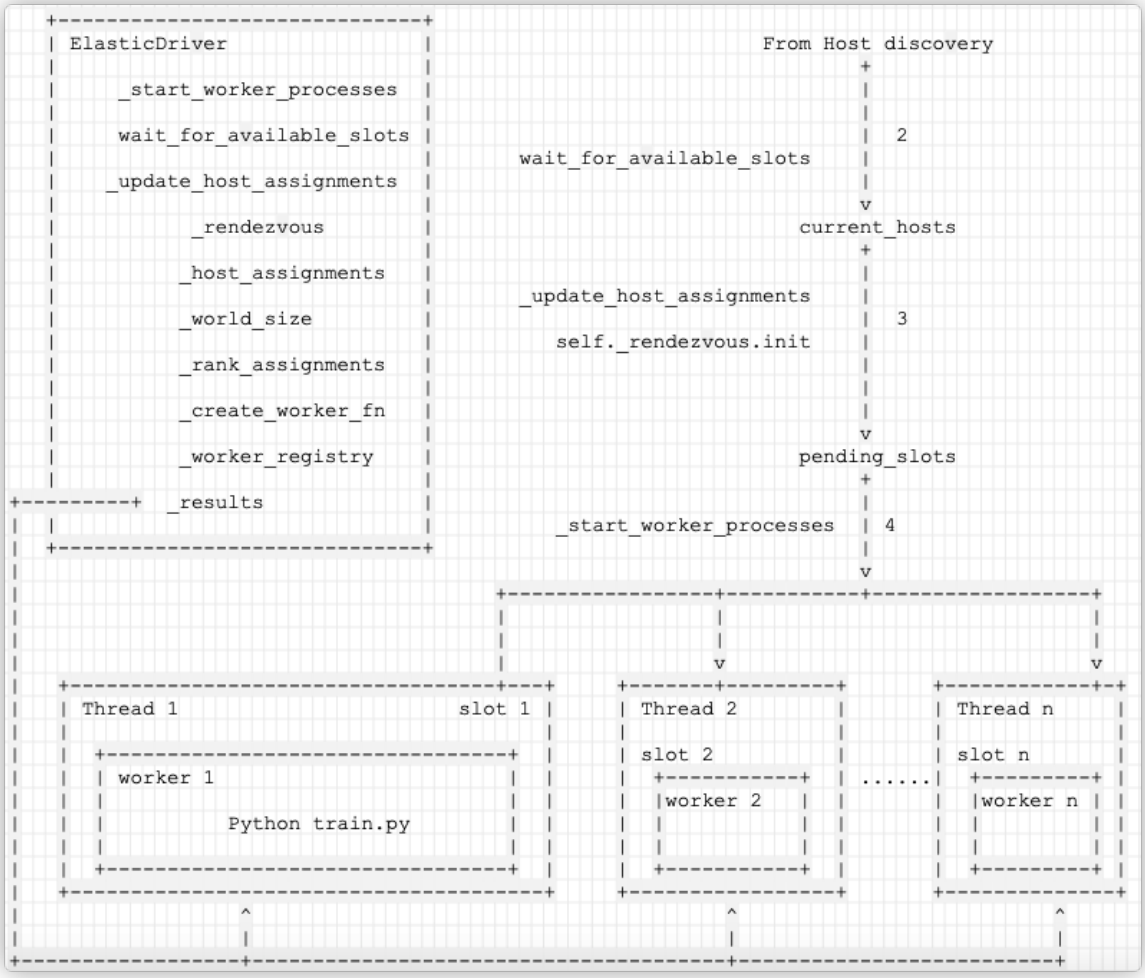
此部分的逻辑层次在后续介绍的 容错机制之上。容错机制是在 Worker 内部，此部分是在 Worker 之上。



6.1 Worker 的逻辑层次

worker 是在 训练脚本之上的阶段，即 Worker 来运行使用 "python train.py" 来运行训练脚本。

所以，上图我们简化如下：



6.2 worker 运行阶段

于是我们提出了新问题如下：

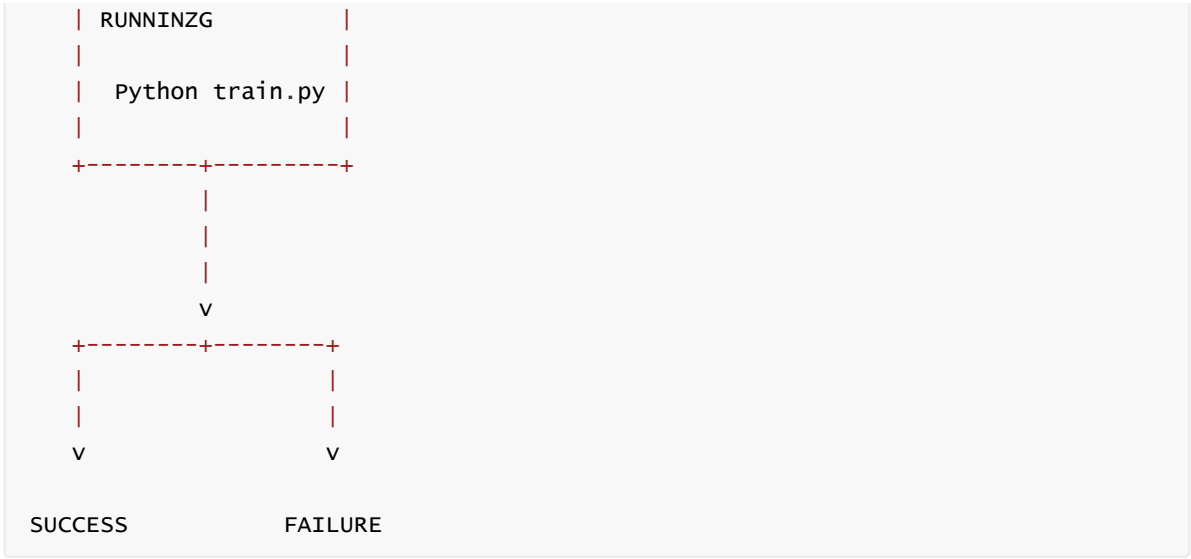
- worker 的运行，怎么才算一个阶段？一共有几个阶段（状态）？
- Driver 根据什么特征来记录 Worker 的运行结果？

从源码中我们可以看到，Worker 有三个状态如下：

```
READY = 'READY'
SUCCESS = 'SUCCESS'
FAILURE = 'FAILURE'
```

所以，Worker 可以分为四个阶段，RUNNING 是我自己加上去的，就是运行训练脚本这个过程，官方没有这个状态，但是我觉得这样应该更清晰。而 SUCCESS 和 FAILURE 就是脚本运行成功与否。

```
          Worker
            +
            |
            |
            |
            v
          READY
            +
            |
            |
            v
   +--------+---------+
```



我们接下来看看运行阶段。

6.2.1 进入 C++ 世界

当 Driver 初始化 / resume（比如收到了HostsUpdateInterrupt ） 时候，就会调用到 hvd.init。

进入 hvd.init 有几个调用途径（按照下面1，2，3 顺序逻辑进行）：

1. 依靠

   ```
   WorkerStateRegistry . _barrier
   ```

   : 作用是当所有 worker 完成之后，会进一步处理。有三个途径会触发这个barrier：

   - start 一个 worker，worker会 hvd.init，进而调用了 gloo in c++，进而 联系 rendezvous，rendezvous 通知 driver，进而在 WorkerStateRegistry 设置自己的状态是 READY，如果达到了 min_np，则会触发了 `_barrier` （**途径 1**）；
   - 新发现一个host，从而导致触发一个 HostsUpdateInterrupt，worker 捕获这个异常之后，进而会 reset，reset 时候会调用 hvd.init，进而和上述一样，最终触发 `_barrier` （**途径 2**）；
   - worker失败，会调用会 `_handle_worker_exit`，进而在 WorkerStateRegistry 设置自己的状态是 FAILURE，会触发了 `_barrier` （**途径 3**）；
2. `_barrier` 继续执行时候，会调用构建时候设置的 handler，即 `_action` 函数，进而调用到了 `_on_workers_recorded`，最终调用到了 `self._driver.resume()`；
3. resume 函数 会 `self._activate_workers(self._min_np)`，最终就是重新生成（也许是部分，根据 pending_slots决定）worker。

6.2.2 构建 Gloo

前文我们提到过，在 python 世界调用 hvd.init 的时候，会进入到 C++世界，这时候如果编译了 GLOO，就建立了一个 GlooContext。

```
namespace {

// All the Horovod state that must be stored globally per-process.
HorovodGlobalState horovod_global;

#if HAVE_GLOO
GlooContext gloo_context;
#endif
```

GlooContext 就得到了一个与 RendezvousServer 通讯的接口。

6.2.2.1 去 Rendezvous 获取信息

从 GlooContext::Initialize 可以知道，需要获取大量配置信息，其中最重要的就是 rank 等信息。

这些信息是在 RendezvousServer 设置存储的。所以 GlooContext 会去 RendezvousServer 进行 http 交互，下面代码还是比较清晰易懂的。

```
#define HOROVOD_GLOO_GET_RANK_AND_SIZE "rank_and_size"

void GlooContext::Initialize(const std::string& gloo_iface) {
  // Create a device for communication
  // TODO(sihan): Add support for multiple interfaces:
  //  https://github.com/facebookincubator/gloo/issues/190
  attr device_attr;
  device_attr.iface = gloo_iface;

  device_attr.ai_family = AF_UNSPEC;
  auto dev = CreateDevice(device_attr);
  auto timeout = GetTimeoutFromEnv();

  auto host_env = std::getenv(HOROVOD_HOSTNAME);
  std::string hostname = host_env != nullptr ? std::string(host_env) :
std::string("localhost");

  int rank = GetIntEnvOrDefault(HOROVOD_RANK, 0);
  int size = GetIntEnvOrDefault(HOROVOD_SIZE, 1);
  int local_rank = GetIntEnvOrDefault(HOROVOD_LOCAL_RANK, 0);
  int local_size = GetIntEnvOrDefault(HOROVOD_LOCAL_SIZE, 1);
  int cross_rank = GetIntEnvOrDefault(HOROVOD_CROSS_RANK, 0);
  int cross_size = GetIntEnvOrDefault(HOROVOD_CROSS_SIZE, 1);

  auto rendezvous_addr_env = std::getenv(HOROVOD_GLOO_RENDEZVOUS_ADDR);
  auto rendezvous_port = GetIntEnvOrDefault(HOROVOD_GLOO_RENDEZVOUS_PORT, -1);

  bool elastic = GetBoolEnvOrDefault(HOROVOD_ELASTIC, false);
  if (elastic && reset_) {
    std::string server_addr = rendezvous_addr_env;
    std::string scope = HOROVOD_GLOO_GET_RANK_AND_SIZE;
    HTTPStore init_store(server_addr, rendezvous_port, scope, rank);

    auto key = hostname + ":" + std::to_string(local_rank);
    std::vector<char> result = init_store.get(key);
    std::string s(result.begin(), result.end());
    std::stringstream ss(s);

    int last_rank = rank;
```

```
    int last_size = size;
    int last_local_rank = local_rank;
    int last_local_size = local_size;
    int last_cross_rank = cross_rank;
    int last_cross_size = cross_size;

    rank = ParseNextInt(ss);

    size = ParseNextInt(ss);
    local_rank = ParseNextInt(ss);
    local_size = ParseNextInt(ss);
    cross_rank = ParseNextInt(ss);
    cross_size = ParseNextInt(ss);

    SetEnv(HOROVOD_RANK, std::to_string(rank).c_str());
    SetEnv(HOROVOD_SIZE, std::to_string(size).c_str());
    SetEnv(HOROVOD_LOCAL_RANK, std::to_string(local_rank).c_str());
    SetEnv(HOROVOD_LOCAL_SIZE, std::to_string(local_size).c_str());
    SetEnv(HOROVOD_CROSS_RANK, std::to_string(cross_rank).c_str());
    SetEnv(HOROVOD_CROSS_SIZE, std::to_string(cross_size).c_str());
  }

  ctx = Rendezvous(HOROVOD_GLOO_GLOBAL_PREFIX,
                   rendezvous_addr_env, rendezvous_port,
                   rank, size, dev, timeout);
  local_ctx = Rendezvous(HOROVOD_GLOO_LOCAL_PREFIX + hostname,
                         rendezvous_addr_env, rendezvous_port,
                         local_rank, local_size, dev, timeout);
  cross_ctx = Rendezvous(HOROVOD_GLOO_CROSS_PREFIX + std::to_string(local_rank),
                         rendezvous_addr_env, rendezvous_port,
                         cross_rank, cross_size, dev, timeout);
}
```

6.2.2.2 RendezvousServer

RendezvousServer 对外提供了 GET 方法。

```
# GET methods
GET_RANK_AND_SIZE = 'rank_and_size'

def _get_value(self, scope, key):
    if scope == GET_RANK_AND_SIZE:
        host, local_rank = key.split(':')
        return self._get_rank_and_size(host, int(local_rank))

    return super(RendezvousHandler, self)._get_value(scope, key)
```

ElasticRendezvousHandler 是 RendezvousServer 的响应handler，其中 `ElasticRendezvousHandler._get_rank_and_size` 函数是：

```
def _get_rank_and_size(self, host, local_rank):
    driver.record_ready(host, local_rank)
    slot_info = driver.get_slot_info(host, local_rank)
    return slot_info.to_response_string().encode('ascii')
```

这里会调用到 driver.record_ready，就是通知 Driver，现在有一个 worker 是 READY 状态。

```
def record_ready(self, host, slot):
    self._worker_registry.record_ready(host, slot)
```

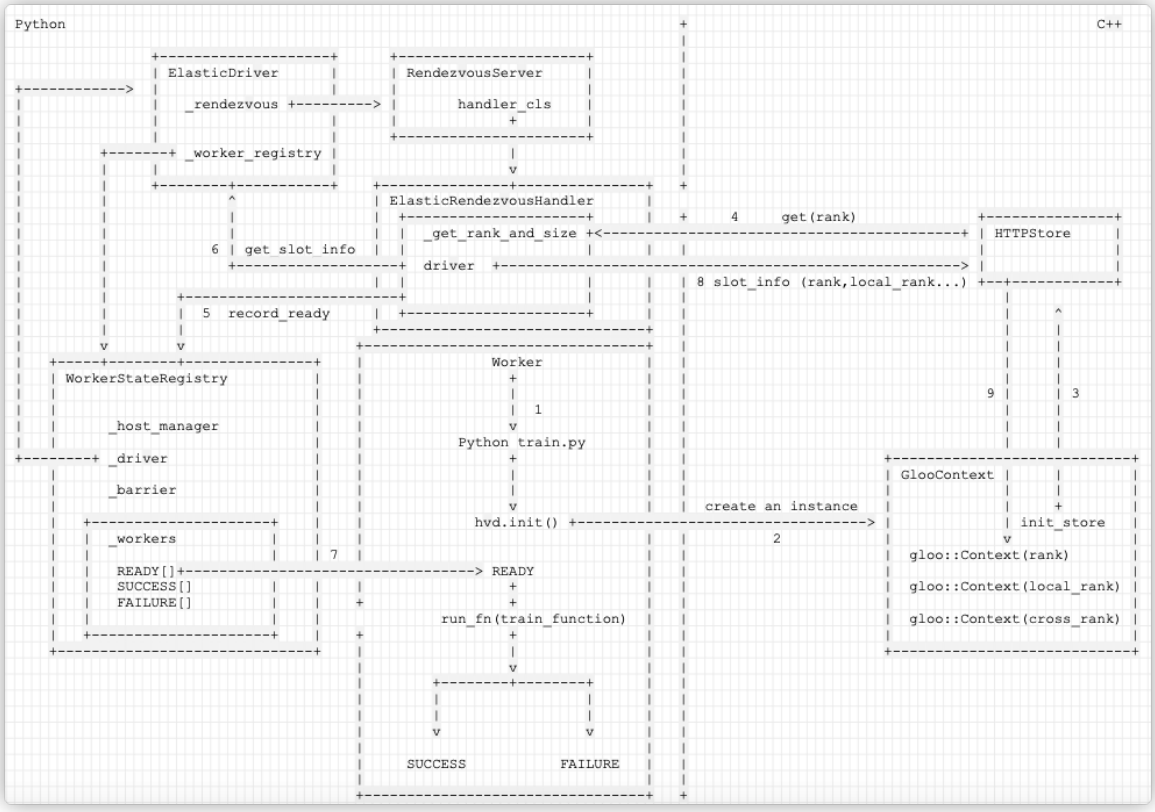
6.2.3 进入READY 状态

当调用 hvd.init -----> GlooContext 建立 之后，会与RendezvousServer 通讯，这之后 Worker 就进入到 READY 状态。

我们需要继续深化下，看看一个 worker 从开始运行到 READY 状态 之间都发生了什么。

1. Worker 开始调用 python train.py；
2. 在 train.py 之中，调用 hvd.init()，此方法会深入到 C++ 世界，从而生成了 GlooContext；
3. GlooContext 之中，会从环境变量之中得到 Rendezvous Server 的ip, port，进而调用 init_store 生成一个 HTTPStore；
4. 调用 `init_store.get(hostname + ":" + std::to_string(local_rank))` 向 Rendezvous Server 发送请求，要求获得本worker 的 rank 对应的 各种配置（local_rank, cross_rank...，因为 Rendezvous Server 可能会重新初始化从而重新分配)；
5. ElasticRendezvousHandler 是 响应函数，其中会 调用 driver.record_ready(host, local_rank) 从而在 WorkerStateRegistry 的 READY 字典中记录下来，worker 2 已经是 READY 了。
6. 会调用 driver.get_slot_info(host, local_rank) 从 driver 获得 slot info；
7. 此时，Worker 的状态就是 READY（其实 Worker 本身没有这个状态，只是 WorkerStateRegistry 有这个状态）；
8. ElasticRendezvousHandler 会返回 slot info 到 worker 的 C++ 世界；
9. 在 worker 的 C++ 世界 之中继续执行，把 slot info 返回给 GlooContext，进行各种设置；

具体逻辑图如下：



至此，Worker 就可以开始运行了。

6.3 WorkerStateRegistry

WorkerStateRegistry 的作用是 注册运行结果，然后进行对应协调。

其主要成员变量是：

- _driver： 用来联系 Driver，因为会调用 driver 来做处理；
- _host_manager： 用来发现 host；
- _workers : 纪录每种状态的worker，状态包括：'READY'，'SUCCESS'，'FAILURE'；
    - ```
      def count(self, state):
          return len(self._workers[state])
      ```
- _states : 纪录 worker 的状态；
- _barrier : 作用是当所有 worker 完成之后，会进一步处理；

具体定义如下：

```
class WorkerStateRegistry(object):
    def __init__(self, driver, host_manager, reset_limit=None, verbose=False):
        self._driver = driver
        self._host_manager = host_manager
        self._reset_limit = reset_limit
        self._reset_count = 0
        self._lock = threading.Lock()
        self._states = {}
        self._workers = defaultdict(set)
        self._barrier = None
        self._rendezvous_id = 0
        self._verbose = verbose
        self._size = 0
```

6.3.1 初始化

WorkerStateRegistry 在 Driver 之中 进行初始化，并且把自己设置为 Driver 的一个成员变量，这样 Driver 就可以方便调用：

```
self._worker_registry = WorkerStateRegistry(self, self._host_manager,
reset_limit=reset_limit)
```

6.3.2 启动

在 master 启动所有 worker 之前，会调用 reset。

```
def _activate_workers(self, min_np):
    current_hosts = self.wait_for_available_slots(min_np)
    pending_slots = self._update_host_assignments(current_hosts)
    self._worker_registry.reset(self.world_size()) # 这里reset
    self._start_worker_processes(pending_slots)
```

reset 函数中有复杂逻辑。

有两个问题：

- 为什么要有 _barrier?
```

原因是：大部分机器学习算法机制是需要当所有 worker（或者若干worker） 完成之后，才会进一步处理，所以需要等待

。

  - 这里 barrier 的参数 parties 具体数值是 self.world_size()，就是说，只有等到barrier 内部计数达到 self.world_size() 时候，就会激发 self._action 函数。
  - 每个worker 结束时候，都会调用到 `_handle_worker_exit`，最终会 `self._barrier.wait()`。
  - 这样，当所有 worker 都结束时候，barrier 会激发 self._action 函数。
- 设置的 _action 起到什么作用？其作用是：根据本次训练结果，进一步控制，决定下一步动作；

代码如下：

```
def reset(self, size):
    with self._lock:
        self._states.clear()
        self._workers.clear()
        self._barrier = threading.Barrier(parties=size, action=self._action)
        self._rendezvous_id += 1
        self._size = size
```

6.3.3 worker 结束

当 worker 结束时候，会回到 Driver 设置的 _handle_worker_exit。根据 exit_code 来决定是调用 success 函数还是 failure 函数。

```
def  _handle_worker_exit(self, slot_info, exit_code, timestamp):
    if not self.has_rank_assignment(slot_info.hostname, slot_info.local_rank):
        return

    if exit_code == 0:
        rendezvous_id = self._worker_registry.record_success(slot_info.hostname,
slot_info.local_rank)
    else:
        rendezvous_id = self._worker_registry.record_failure(slot_info.hostname,
slot_info.local_rank)

    if self.finished() and self._worker_registry.last_rendezvous() ==
rendezvous_id:
        name = '{}[{}]'.format(slot_info.hostname, slot_info.local_rank)
        self._results.add_result(name, (exit_code, timestamp))
```

从而调用到 WorkerStateRegistry 之中。

```
def record_ready(self, host, slot):
    return self._record_state(host, slot, READY)

def record_success(self, host, slot):
    return self._record_state(host, slot, SUCCESS)

def record_failure(self, host, slot):
    return self._record_state(host, slot, FAILURE)
```

而 `_record_state` 函数会使用 `self._workers[state].add(key)` 来纪录状态，并且调用 _wait。

```
def _record_state(self, host, slot, state):
    if self._driver.finished():
        return self._rendezvous_id
    if self._host_manager.is_blacklisted(host):
        return self._rendezvous_id

    key = (host, slot)
    with self._lock:
        if key in self._states:
            if state == FAILURE:
                self._barrier.reset()

        if key not in self._states or state == FAILURE:
            self._states[key] = state
            self._workers[state].add(key)

        rendezvous_id = self._rendezvous_id

    rendezvous_id = self._wait(key, state, rendezvous_id)
    return rendezvous_id
```

`_wait` 会并且调用 `self._barrier.wait()` 来等待，这是为了等待其他 worker 的信息，最后一起处理。

```
def _wait(self, key, state, rendezvous_id):
    while True:
        try:
            self._barrier.wait()
            return rendezvous_id
        except threading.BrokenBarrierError:
            if self._barrier.broken:
                # Timeout or other non-recoverable error, so exit
                raise

            # Barrier has been reset
            with self._lock:
                # Check to make sure the reset was not caused by a change of
state for this key
                rendezvous_id = self._rendezvous_id
                saved_state = self._states.get(key, state)
                if saved_state != state:
                    # This worker changed its state, so do not attempt to wait
again to avoid double-counting
                    raise RuntimeError('State {} overridden by {}'.format(state,
saved_state))
```

6.3.4 进一步控制

_action 函数会在所有worker 结束之后，进行判断，控制。

```
def _action(self):
    self._on_workers_recorded()
```

_on_workers_recorded 函数会完成控制逻辑。

- 判断是否有一个 worker 成功，如果有一个 worker 成功了，就关闭其他process，结束训练；因为此时所有 worker 都已经运行结束，所以只要有一个 worker 成功，就可以跳出循环；
- 如果所有的 worker 都失败了，就结束训练；
- 把失败的 worker 纪录到 黑名单；
- 如果所有的 host 都在黑名单，则结束训练；
- 如果已经到了最大重试数目，则结束训练；
- 否则调用 _driver.resume() 重启训练，因为已经 commit 了，所以会自动恢复训练；

具体代码如下：

```
def _on_workers_recorded(self):
    # Check for success state, if any process succeeded, shutdown all other
processes
    if self.count(SUCCESS) > 0:
        self._driver.stop()
        return

    # Check that all processes failed, indicating that processing should stop
    if self.count(FAILURE) == self._size:
        self._driver.stop()
        return

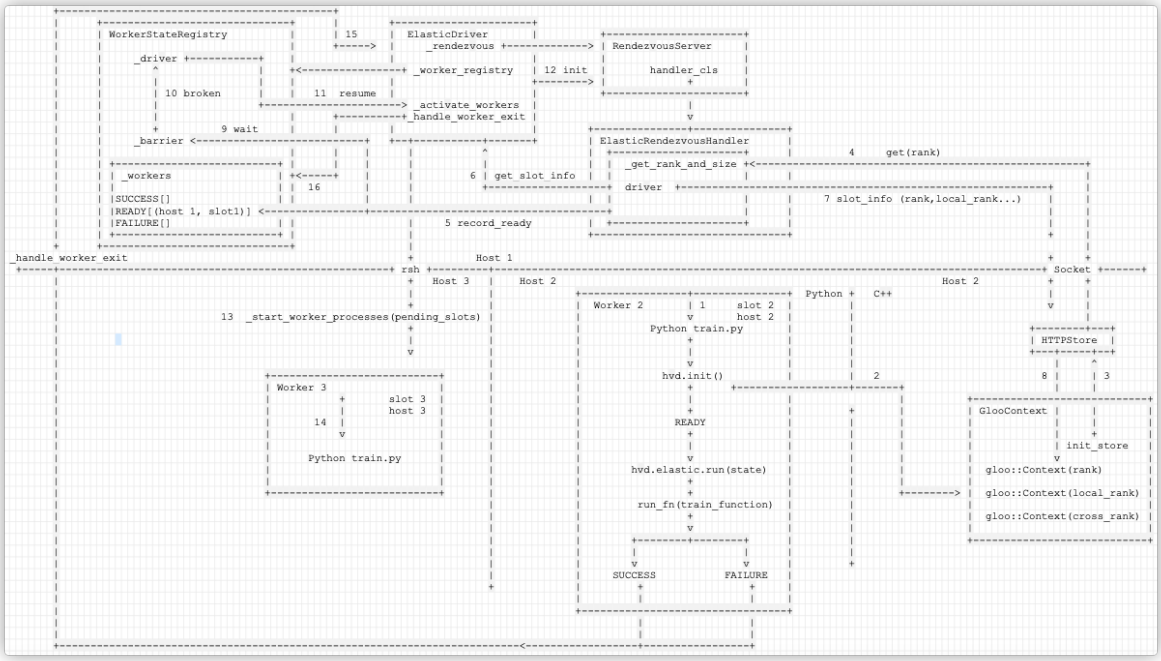
    # Check for failures, and add them to the blacklisted hosts list
    failures = self.get(FAILURE)
    for host, slot in failures:
        self._host_manager.blacklist(host)

    # If every active host is blacklisted, then treat this as job failure
    if all([self._host_manager.is_blacklisted(host) for host, slot in
self.get_recorded_slots()]):
        self._driver.stop()
        return

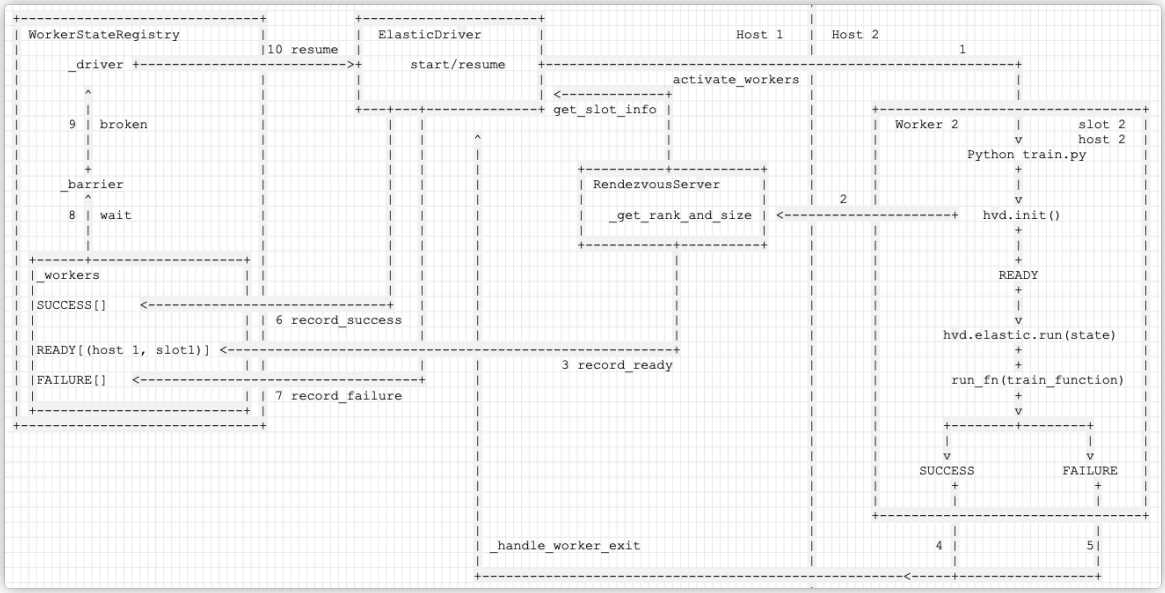
    # Check that we have already reset the maximum number of allowed times
    if self._reset_limit is not None and self._reset_count >= self._reset_limit:

self._driver.stop(error_message=constants.RESET_LIMIT_EXCEEDED_MESSAGE.format(se
lf._reset_limit))
        return

    try:
        self._reset_count += 1
        self._driver.resume()
    except Exception:
        self._driver.stop()
```

6.4 Driver.resume 场景

resume 的作用 就是 所有都重新来过。

```
def resume(self):
    self._activate_workers(self._min_np)
```

我们之前分析的场景是：一个 worker 从开始运行到 READY 状态 之间都发生了什么。

现在，我们加上一个情形：就是当 Driver 在 resume 的时候，发现居然有新节点，随即启动了一个新 worker 3。

1. Worker 2 开始调用 python train.py；
2. 在 train.py 之中，调用 hvd.init()，此方法会深入到 C++ 世界，从而生成了 GlooContext；
3. GlooContext 之中，会从环境变量之中得到 Rendezvous Server 的ip, port，进而调用 init_store 生成一个 HTTPStore；
4. 调用 `init_store.get(hostname + ":" + std::to_string(local_rank))` 向 Rendezvous Server 发送请求，要求获得本worker 的 rank 对应的 各种配置（local_rank, cross_rank...，因为 Rendezvous Server 可能会重新初始化从而重新分配)；
5. ElasticRendezvousHandler 是 响应函数，其中会 调用 driver.record_ready(host, local_rank) 从而在 WorkerStateRegistry 的 READY 字典中记录下来，worker 2 已经是 READY 了。
6. 会调用 driver.get_slot_info(host, local_rank) 从 driver 获得 slot info；
7. 把 slot info 返回给 worker 中的 Http_store；
8. 在 worker 2 之中继续执行，把 slot info 返回给 GlooContext，进行各种设置；
9. 我们接着 第 5 项继续进行；`record_ready` 之中 会调用 `rendezvous_id = self._wait(key, state, rendezvous_id)` 来在 `WorkerStateRegistry . _barrier` 之上等待； `_barrier` 的类型是`threading.Barrier(parties=size, action=self._action)` ，
10. 如果 READY 的 worker 数目达到了 Horovod 设置的 min-np，就是可以启动的最小 worker 数目， `_barrier` 就结束使命，就 broken，继续执行；
11. `_barrier` 继续执行时候，会调用构建时候设置的 handler，即 `_action` 函数，进而调用到了 `_on_workers_recorded`，最终调用到了 `self._driver.resume()`；
12. ElasticDriver 的 resume 函数调用到了 `_activate_workers`，其定义如下，可以看到，如果此时 discovery 脚本已经发现了新节点，进而返回了 `pending_slots`，`pending_slots` 就是可以在这些 slot 之上启动新 worker 的，于是 就会 调用 `_start_worker_processes`：

    1. ```
       def _activate_workers(self, min_np):
           current_hosts = self.wait_for_available_slots(min_np)
           pending_slots = self._update_host_assignments(current_hosts)
           self._worker_registry.reset(self.world_size())
           self._start_worker_processes(pending_slots)
       ```

13. _start_worker_processes 会开启一个新的 worker : Worker 3；
14. Worker 3 也执行 python train.py，至此，新worker启动完毕；
15. 回到worker 2，如果训练结束，则会依据训练结果，返回 SUCCESS 或者 FAILURE 到 Driver；
16. Driver 会调用 _handle_worker_exit 对训练结果进行处理；

至此，新的逻辑完成。
       ```



我们可以扩大部分细节看看，就是Driver + 一个 Worker，这样看得更清楚：



至此，worker部分分析完毕，下一篇我们看看如何处理错误。