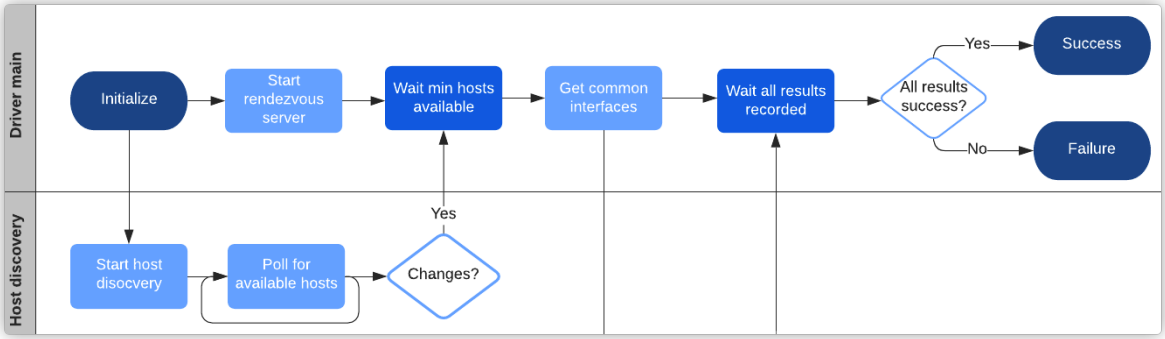
# 弹性训练之 Driver

目录



## 0x00 摘要

看看 horovod 弹性实现中 的 Driver 角色。本部分对应架构图中的 Driver main 部分，因为这部分和 Host discovery 强相关，所以一起展示出来。因为弹性训练的主体是 Driver，所以本文是把 调用代码和 Driver 一起分析。




## 0x01 角色

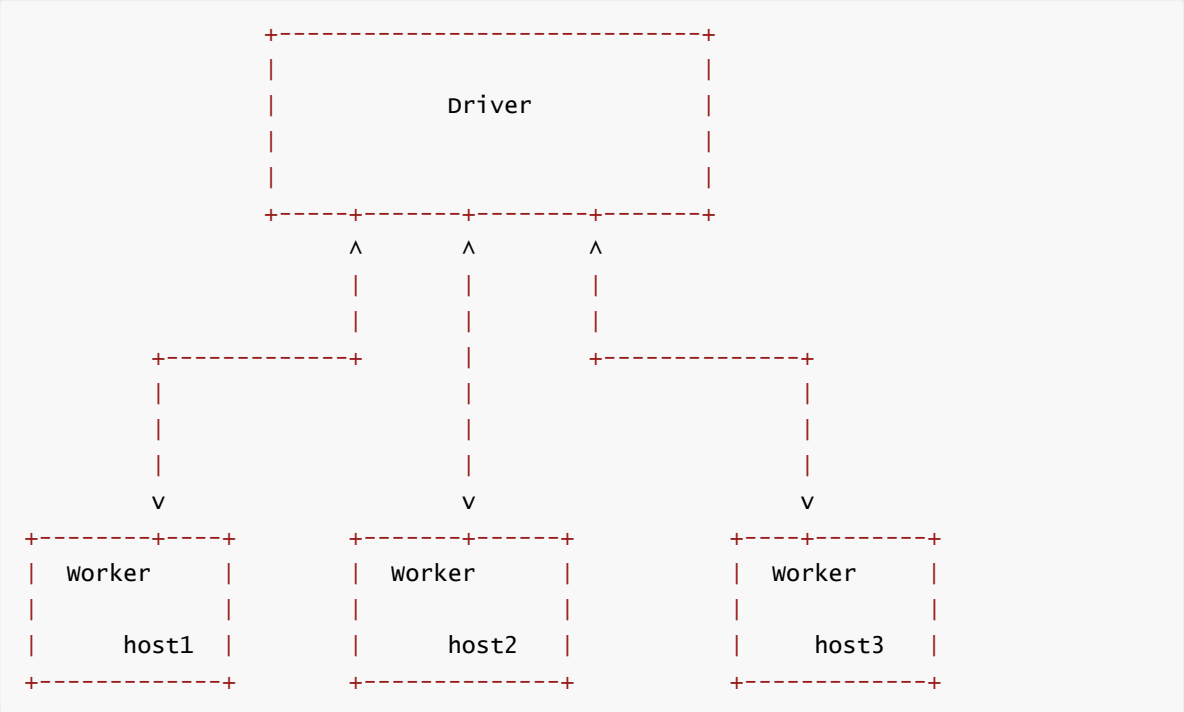
我们首先要回忆一下弹性训练中的角色设定。

## 1.1 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。

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（控制节点）职责：

- 负责实时检测现有 worker（工作节点）是否有变化，掉线情况；
- 负责通过脚本来实时监控 host 是否有变化；
- 负责分配任务到存活的worker（工作节点）；
- 在有AllReduce 调用失败导致进程失败的情况下，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，则在新host之上生成新的 worker，新 worker 和 旧 worker 一起构造成一个新的通信环。

worker（工作节点）职责：

- 负责汇报（其实是被动的，没有主动机制）自己的状态（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。

# 0x02 调用部分

我们首先分析调用部分，弹性调用具体是从普适到特殊，一点点深入。

## 2.1 _run

前文介绍了 horovod 程序的入口是 _run 函数。可以看到，会依据是否是弹性训练来选择不同的路径。我们本文开始介绍 _run_elastic。

```
def _run(args):
    # if hosts are not specified, either parse from hostfile, or default as
    # localhost
    if not args.hosts and not args.host_discovery_script:
        if args.hostfile:
            args.hosts = hosts.parse_host_files(args.hostfile)
        else:
            # Set hosts to localhost if not specified
            args.hosts = 'localhost:{np}'.format(np=args.np)

    # Convert nics into set
    args.nics = set(args.nics.split(',')) if args.nics else None

    if _is_elastic(args):
        return _run_elastic(args) # 本文在这里
    else:
        return _run_static(args)
```

## 2.2 _run_elastic

此部分逻辑如下：

- 首先，如果参数配置了“获取参数的脚本”，则调用 discovery.HostDiscoveryScript 得到一个 object（目前只是一个object，未来在构建 ElasticDriver 的时候会获取 host 信息）。否则就直接读取固定 host 配置；
- 其次，使用 host 配置以及其他信息来配置 ElasticSettings；
- 最后，调用 gloo_run_elastic 来进行弹性训练；

代码如下：

```
def _run_elastic(args):
    # construct host discovery component
    if args.host_discovery_script:
        discover_hosts =
discovery.HostDiscoveryScript(args.host_discovery_script, args.slots)
    elif args.hosts:
        _, available_host_slots = hosts.parse_hosts_and_slots(args.hosts)
        discover_hosts = discovery.FixedHosts(available_host_slots)
    ......

    # horovodrun has to finish all the checks before this timeout runs out.
    settings = elastic_settings.ElasticSettings(discovery=discover_hosts,
                                                min_np=args.min_np or args.np,
                                                max_np=args.max_np,

 elastic_timeout=args.elastic_timeout,
                                                reset_limit=args.reset_limit,
                                                num_proc=args.np,
                                                verbose=2 if args.verbose else
0,
                                                ssh_port=args.ssh_port,
```

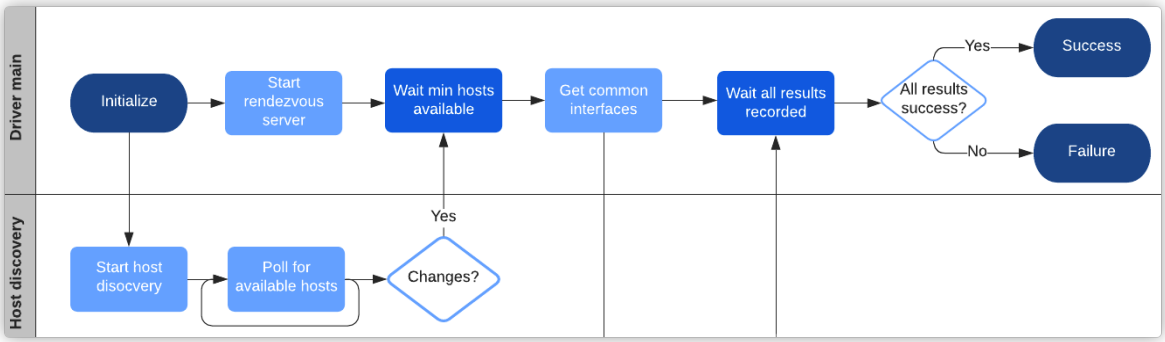
```
 ssh_identity_file=args.ssh_identity_file,
                                                 extra_mpi_args=args.mpi_args,
                                                 key=secret.make_secret_key(),
                                                 start_timeout=tmout,

 output_filename=args.output_filename,
                                                 run_func_mode=args.run_func is
not None,
                                                 nics=args.nics,...)

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

## 2.3 gloo_run_elastic

这部分开始，就是架构图中第一部分：




主要做了如下：

- 定义了 get_common_interfaces，这是一个可以获取网络路由信息以及host能力的函数；需要注意的一点是：会等待所需的最小节点数目，然后才会开始获取网络路由。
- _exec_command_fn 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；
- 建立了一个 RendezvousServer，用来保存各种host信息；
- 使用以上这些参数和 command 参数来运行 launch_gloo_elastic，command参数就是类似 `python train.py`；

```
def gloo_run_elastic(settings, env, command):

    def get_common_interfaces(driver):
        # Host-to-host common interface detection requires at least 2 hosts in
an elastic job.
        min_hosts = _get_min_start_hosts(settings)
        current_hosts = driver.wait_for_available_slots(settings.num_proc,
min_hosts=min_hosts)
        return driver_service.get_common_interfaces(settings,
current_hosts.host_assignment_order)

    exec_command = _exec_command_fn(settings)
    rendezvous = RendezvousServer(settings.verbose)
    launch_gloo_elastic(command, exec_command, settings, env,
get_common_interfaces, rendezvous)
```

## 2.4 get_common_interfaces

get_common_interfaces 可以获取网络路由信息以及host。

- 如果配置了远端host，则会分布式执行，在各个 host 之上执行，获取每个 host 的网卡和路由信息。
- 否则就获取本地网卡等信息。

具体函数在：runner/driver/driver_service.py

```
def get_common_interfaces(settings, all_host_names, remote_host_names=None,
fn_cache=None):
    if remote_host_names is None:
        remote_host_names = network.filter_local_addresses(all_host_names)

    if len(remote_host_names) > 0:
        if settings.nics:
            # If args.nics is provided, we will use those interfaces. All the
workers
            # must have at least one of those interfaces available.
            nics = settings.nics
        else:
            # Find the set of common, routed interfaces on all the hosts (remote
            # and local) and specify it in the args to be used by NCCL. It is
            # expected that the following function will find at least one
interface
            # otherwise, it will raise an exception.
            local_host_names = set(all_host_names) - set(remote_host_names)
            nics = _driver_fn(all_host_names, local_host_names, settings,
fn_cache=fn_cache)

    else:
        nics = get_local_interfaces(settings)
    return nics
```

get_local_interfaces 是获取本地 host 的网卡信息。

```
def get_local_interfaces(settings):
    if settings.verbose >= 2:
        print('All hosts are local, finding the interfaces '
              'with address 127.0.0.1')
    # If all the given hosts are local, find the interfaces with address
    # 127.0.0.1
    nics = set()
    for iface, addrs in net_if_addrs().items():
        if settings.nics and iface not in settings.nics:
            continue
        for addr in addrs:
            if addr.family == AF_INET and addr.address == '127.0.0.1':
                nics.add(iface)
                break

    return nics
```

## 2.5 获取异地网卡信息

此处信息在前文中已经讲述，我们精简如下：

_driver_fn 的作用是分布式执行 探寻函数，作用是：

- 启动 service 服务；
- 使用 driver.addresses() 获取 Driver 服务的地址（使用 `self._addresses = self._get_local_addresses()` 完成）；
- 使用 _launch_task_servers（利用 Driver 服务的地址）在每个 worker 之中启动 task 服务，然后 task 服务会在 service 服务中注册；
- 因为是一个环形，每个 worker 会探测 worker index + 1 的所有网络接口；
- 最后 _run_probe 返回一个所有 workers 上的所有路由接口的交集；

```
@cache.use_cache()
def _driver_fn(all_host_names, local_host_names, settings):
    """
    launches the service service, launches the task service on each worker and
    have them register with the service service. Each worker probes all the
    interfaces of the worker index + 1 (in a ring manner) and only keeps the
    routed interfaces. Function returns the intersection of the set of all the
    routed interfaces on all the workers.
    :param all_host_names: list of addresses. for example,
        ['worker-0','worker-1']
        ['10.11.11.11', '10.11.11.12']
    :type all_host_names: list(string)
    :param local_host_names: host names that resolve into a local addresses.
    :type local_host_names: set
    :param settings: the object that contains the setting for running horovod
    :type settings: horovod.runner.common.util.settings.Settings
    :return: example: ['eth0', 'eth1']
    :rtype: list[string]
    """
    # Launch a TCP server called service service on the host running horovod
    num_hosts = len(all_host_names)
    driver = HorovodRunDriverService(num_hosts, settings.key, settings.nics)

    # Have all the workers register themselves with the service service.
    _launch_task_servers(all_host_names, local_host_names,
                         driver.addresses(), settings)
    try:
        return _run_probe(driver, settings, num_hosts)
    finally:
        driver.shutdown()
```

## 2.6 launch_gloo_elastic

到了这里，才是正式调用起 gloo 弹性系统，就是生成 Driver 相关部分 & 建立 弹性训练的 worker。

运行之中，只有一个 RendezvousServer，launch_gloo_elastic 也只运行一次。

逻辑如下：

- 如果需要配置输出文件，则创建；
- 使用"发现脚本"等作为参数 建立 ElasticDriver；
- 使用 create_rendezvous_handler 作为 handler 来启动 RendezvousServer；
- 使用 driver.wait_for_available_slots 来等待所需的最小数目 slots；

- 如果等到了，就调用 get_common_interfaces 获取网络路由等，从而得到 server ip；
- 注册 shutdown event；
- 利用 get_run_command 得到运行的命令；
- 利用 _create_elastic_worker_fn 建立 弹性训练的 worker；

简单代码如下：

```
def launch_gloo_elastic(command, exec_command, settings, env,
get_common_interfaces, rendezvous):
    # Make the output directory if it does not exist
    if settings.output_filename:
        _mkdir_p(settings.output_filename)

    # 使用"发现脚本"等作为参数 建立 ElasticDriver
    driver = ElasticDriver(rendezvous, settings.discovery,
                           settings.min_np, settings.max_np,
                           timeout=settings.elastic_timeout,
                           reset_limit=settings.reset_limit,
                           verbose=settings.verbose)

    handler = create_rendezvous_handler(driver)
    global_rendezv_port = rendezvous.start(handler) # 启动 RendezvousServer
    driver.wait_for_available_slots(settings.num_proc)

    nics = get_common_interfaces(driver) # 获取网络路由等
    server_ip = network.get_driver_ip(nics)

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

    # 建立 弹性训练的 worker
    create_worker = _create_elastic_worker_fn(exec_command, run_command, env,
event)

    driver.start(settings.num_proc, create_worker)
    res = driver.get_results()
    driver.stop()

    for name, value in sorted(res.worker_results.items(), key=lambda item:
item[1][1]):
        exit_code, timestamp = value
```

这里很复杂，我们需要逐一分析。

首先，我们看看 get_run_command，这个在前文spark gloo之中介绍过，这里再说一下。

它会调用 create_run_env_vars 得到gloo需要信息，并据此构建 run_command，其格式如下：

```
HOROVOD_GLOO_RENDEZVOUS_ADDR=1.1.1.1 HOROVOD_GLOO_RENDEZVOUS_PORT=2222
HOROVOD_CPU_OPERATIONS=gloo HOROVOD_GLOO_IFACE=lo HOROVOD_CONTROLLER=gloo python
```

可以看到，elastic 和 spark gloo 版本很类似，都是使用 RendezvousServer 来完成一些master的控制功能。

其次，我们看看Driver主体。

# 0x03 Driver Main

## 3.1 ElasticDriver

定义如下，基本成员是：

- _rendezvous： driver 会根据当前正在运行的节点重新执行一个 RendezvousServer，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；
- _host_manager： HostManager 负责发现，管理各种 host；
- _worker_registry： WorkerStateRegistry
- _discovery_thread： 负责后台定期探寻 host，具体会调用 _host_manager 完成功能；
- _worker_clients： WorkerNotificationClient，每一个 worker 对应一个；
- _host_assignments： host 分配信息；
- _rank_assignments： rank 分配信息。rank 可以认为是代表分布式任务里的一个执行训练的进程。Rank 0 在Horovod中通常具有特殊的意义：它是负责此同步的设备。
- _world_size： 进程总数量，会等到所有world_size个进程就绪之后才会开始训练；
- _wait_hosts_cond： 类型是 threading.Condition，目的是等待 训练所需最小的 host 数目；

具体定义如下：

```
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)

        self._host_assignments = {}
        self._rank_assignments = {}
        self._world_size = 0

        self._wait_hosts_cond = threading.Condition()

        self._create_worker_fn = None
        self._worker_clients = {}

        self._worker_registry = WorkerStateRegistry(self, self._host_manager,
reset_limit=reset_limit)
        self._results = ResultsRecorder()
        self._shutdown = threading.Event()

        self._discovery_thread = threading.Thread(target=self._discover_hosts)
        self._discovery_thread.daemon = True
        self._discovery_thread.start()
```

创建Driver之后，接下来的主要动作是：

- 等待最小数目 host。
- 配置 worker。
- 启动 driver，其内部会启动worker。
- Driver 等待 worker 的运行结果。

我们逐步分析。

## 3.2 等待最小数目 host

启动之后，会调用 `driver.wait_for_available_slots(settings.num_proc)` 等待最小数目host。

可以看到，这里就是无限循环等待，如果 `avail_slots >= min_np and avail_hosts >= min_hosts` 才会返回。其实，就是看 self._host_manager.current_hosts 的数目是否已经达到了 所需最小的 host 数目，而且 slot 也达到了所需最小数目。

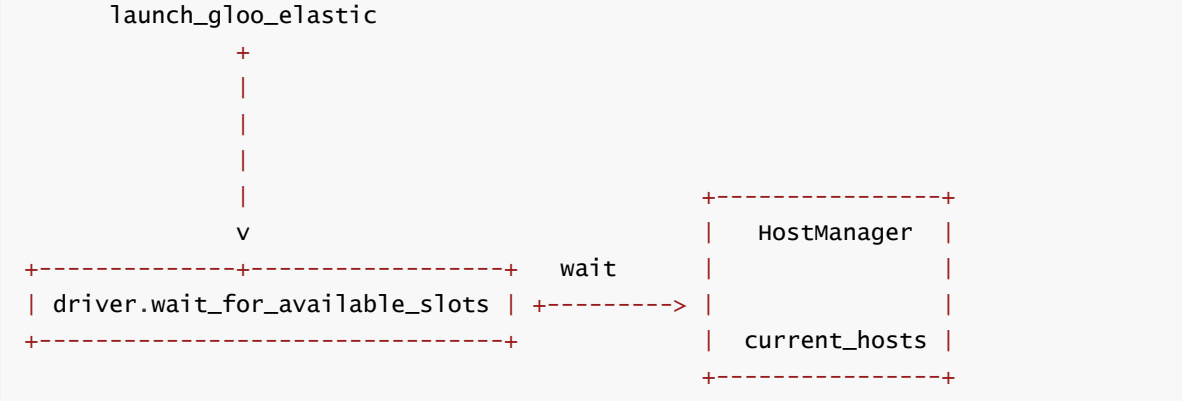
```
def wait_for_available_slots(self, min_np, min_hosts=1):
    tmout = timeout.Timeout(self._timeout,  message='')
    self._wait_hosts_cond.acquire()

    try:
        while True: # 无限循环等待
            current_hosts = self._host_manager.current_hosts

            avail_slots = current_hosts.count_available_slots()
            avail_hosts = len(current_hosts.available_hosts)

            if avail_slots >= min_np and avail_hosts >= min_hosts:
                return current_hosts
            if self._shutdown.is_set():
                raise RuntimeError('Job has been shutdown, see above error
messages for details.')
            self._wait_hosts_cond.wait(tmout.remaining())
            tmout.check_time_out_for('minimum number of slots to become
available')
    finally:
        self._wait_hosts_cond.release()
```

逻辑如下：




## 3.3 配置 worker

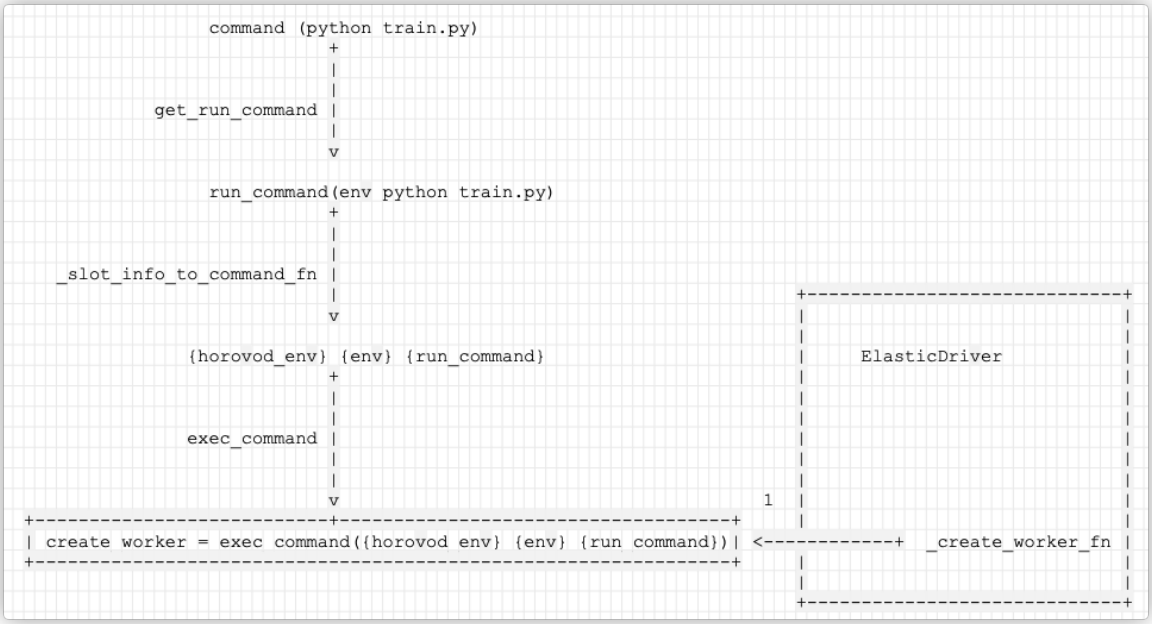
配置过程 是由 _create_elastic_worker_fn 完成。

_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；

这几个概念关系具体如下：




## 3.4 启动 driver

```
driver.start(settings.num_proc, create_worker)
```

具体启动经历了以下几个步骤。

### 3.4.1 start

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

### 3.4.2 _activate_workers

ElasticDriver 的 resume / start 函数会调用到 `_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`：

```
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)
```

### 3.4.3 _start_worker_processes

启动之后，在一个线程中通过run_worker启动worker，然后使用 self._results.expect(thread) 向 ResultsRecorder 放入 worker 线程。这是等待结果的关键。

```
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) # 启动训练线程
        thread.daemon = True
        thread.start()
        self._results.expect(thread) # 等待运行结果
```

## 3.5 等待运行结果

Driver 使用 如下得到结果。

```
def get_results(self):
    return self._results.get_results()
```

_results是 ResultsRecorder 类型，所以我们需要看看其实现。

### 3.5.1 ResultsRecorder

几个功能如下：

- expect 等待 thread：采用 expect 来 self._worker_threads.put(worker_thread)，这样就知道应该等待哪些 thread。
- add_result 添加结果：`_handle_worker_exit` 会 在 record 之后，调用 `self._results.add_result(name, (exit_code, timestamp))` 纪录结果；
- get_results 获取结果：driver 就是调用此函数，获取结果，利用了 join。

```
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)

    def add_result(self, key, value):
        if key in self._worker_results:
            return
        self._worker_results[key] = value

    def get_results(self):
        while not self._worker_threads.empty():
            worker_thread = self._worker_threads.get()
            worker_thread.join()
        return Results(self._error_message, self._worker_results)
```

### 3.5.2 worker 结束

Driver 使用 _handle_worker_exit 来等待具体 worker结束。根据 worker 的返回来决定如何处理。

`_handle_worker_exit` 是运行在 worker thread 之中，运行时候，会通过 `self._results.add_result` 往 ResultsRecorder 注册信息。

```
def _handle_worker_exit(self, slot_info, exit_code, timestamp):
    if not self.has_rank_assignment(slot_info.hostname, slot_info.local_rank):
        # Ignore hosts that are not assigned a rank
        return

    if exit_code == 0: # 顺利完成记录
        rendezvous_id = self._worker_registry.record_success(slot_info.hostname,
slot_info.local_rank)
    else: # 否则记录失败
        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)) # 往
ResultsRecorder注册信息
```

具体如下：