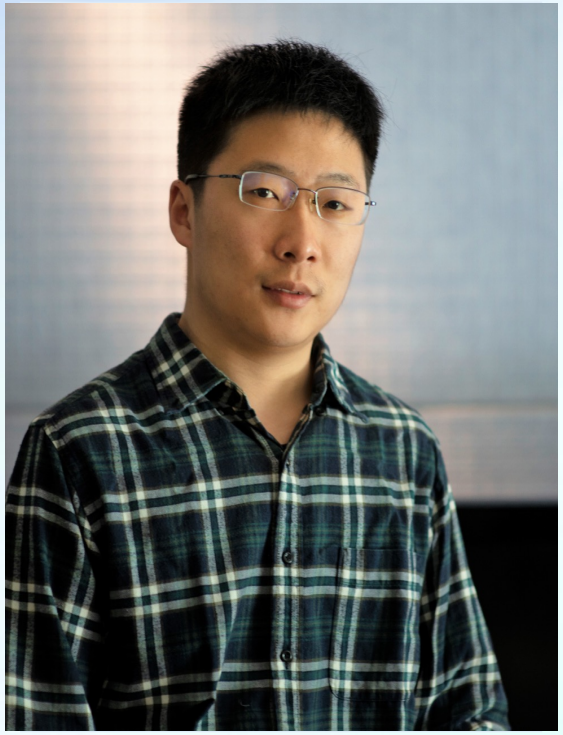
分享主题：DLRover on Ray

面向深度学习的异构融合引擎的设计与实现

陈 天 熠（天 奕）

2025/12/20

v1.1



陈天熠

职位：AI Infra研发工程师

个人简介：19年入职蚂蚁，开始从事分布式计算引擎的研发工作。构建了基于Ray的面向流式训练的计算引擎，支撑了包括支付宝首页推荐相关的在线学习链路。现主要负责DLRover项目的研发，支持蚂蚁的大模型训练。

蚂蚁开源 ANT OPEN SOURCE

RAY

CONTENT

目录

认识DLRover

初识

项目地址：https://github.com/intelligent-machine-learning/dlrover

- 形式：主体开源+内部增量开发
- 目的：降低深度学习训练门槛，低成本方式优化运行时实现

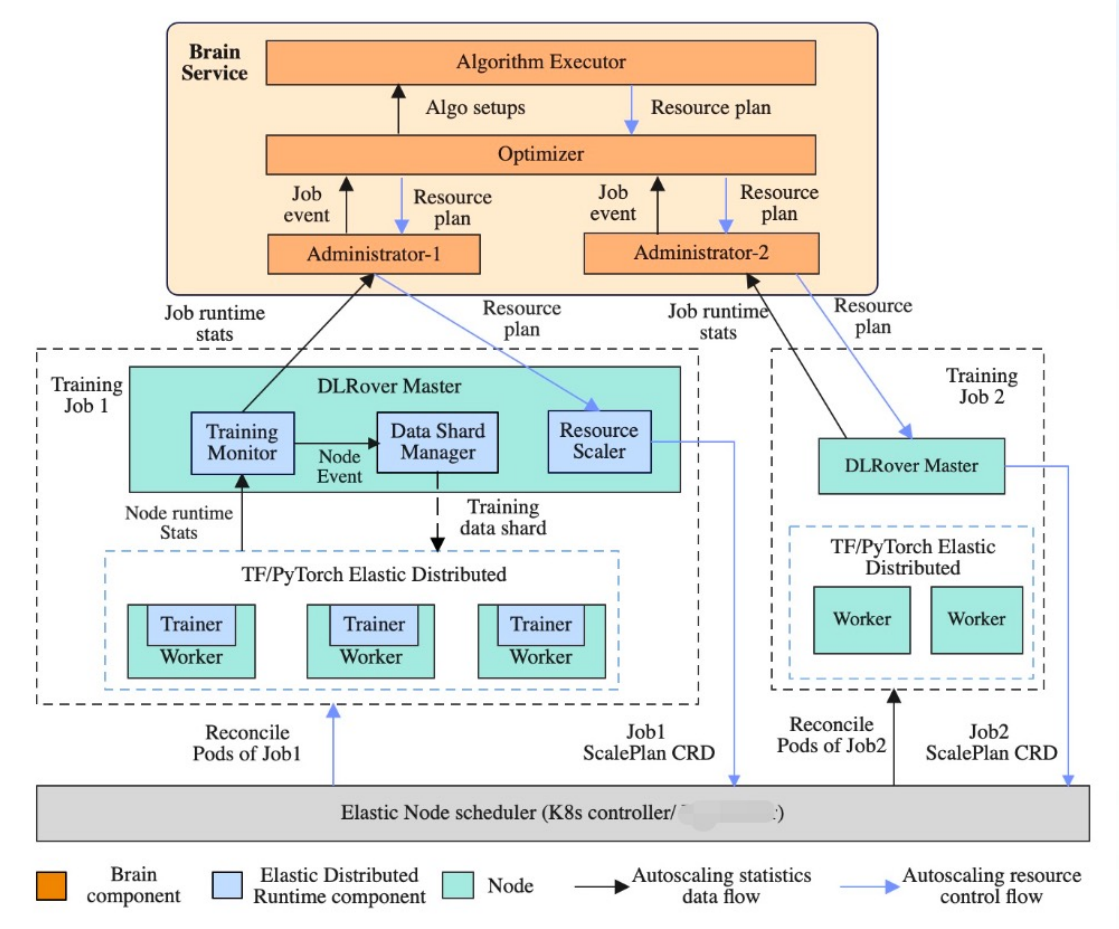
DLRover: An Automatic Distributed Deep Learning System

DLRover makes the distributed training of large AI models easy, stable, fast and green. It can automatically train the Deep Learning model on the distributed cluster. It helps model developers to focus on model arichtecture, without taking care of any engineering stuff, say, hardware acceleration, distributed running, etc. Now, it provides automated operation and maintenance for deep learning training jobs on K8s/Ray. Major features as

认识DLRover

基于K8S的架构和应用场景

- TensorFlow：
  - 场景：搜推/广告
  - 规模：离线20w+core
  - 功能：
    - 生命周期管理
    - 运行时容错
    - 运行时资源调优
- Torch：
  - 场景：大模型
  - 规模：万卡（gpu/xpu）
  - 功能：
    - 生命周期管理
    - 运行前自检
    - 运行时容错
    - 运行后诊断



现有架构和挑战

大模型场景——用户侧：

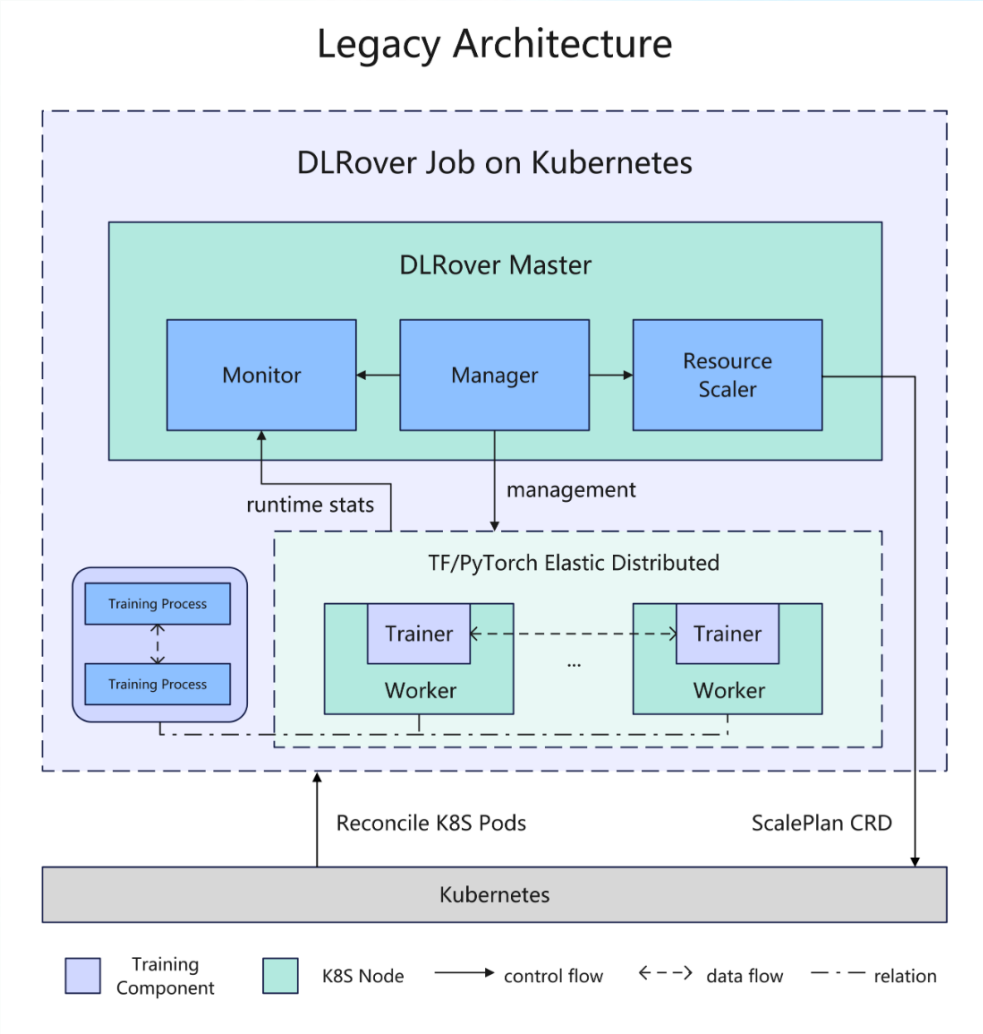
torchrun -arg0 -arg1 train.py

核心实现：继承并扩展了Torch的ElasticAgent

1. Worker侧使用Torch的ElasticAgent管理进程
2. Master侧重写包括节点生命周期，组网等实现

⇒ dlrover-run -arg0 -arg1 train.py

一个运行时全面增强的torchrun实现



现有架构和挑战

能力具备：

超大规模下执行一个同构的大模型训练计算

技术关键点：

1. Master节点的核心领域模型是机器节点，即DLRover管控的是机器的生命周期
2. 复用了Torch的ElasticAgent实现，使用ElasticAgent的接口管理子训练进程
3. Master和Worker之间采用单向交互设计（worker->master by grpc/http）

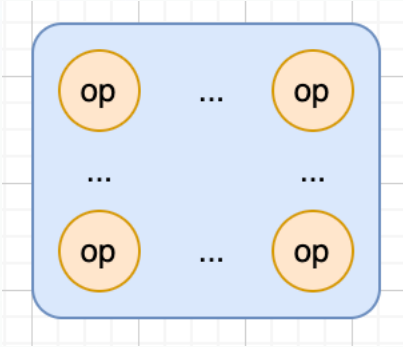
挑战：

1. 如果需要一个非torch弹性训练组的实例，比如实现一个vllm的负载，一个cpu的负载？即如何支持异构的计算？
2. Master节点的核心领域模型是机器节点，无法完成进程级的细粒度操作。异构场景下，更是强需。
3. 异构场景下，跨进程通信如何定义和实现？能否改进为双向通信？按现有grpc实现方式成本太高

结论：不够灵活

现有架构和挑战

基于Torch的SPMD计算

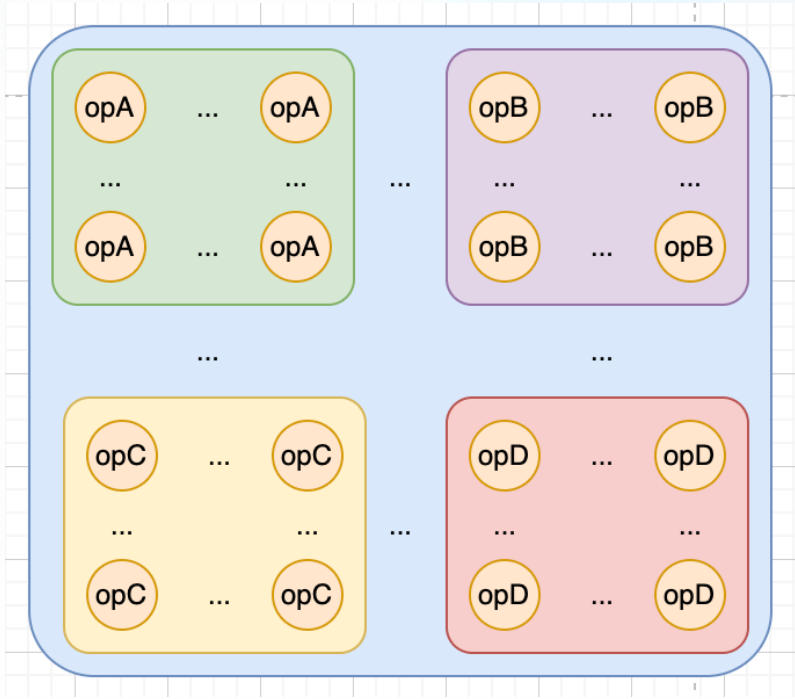


结论：不够灵活

本质

目标

更一般的MPMD计算



继承现有能力，且支持更多场景的灵活架构设计

On Ray的设计与实现

Kubernetes -> Ray

选用 Ray作为新的分布式底盘，主要考虑以下几个原因：

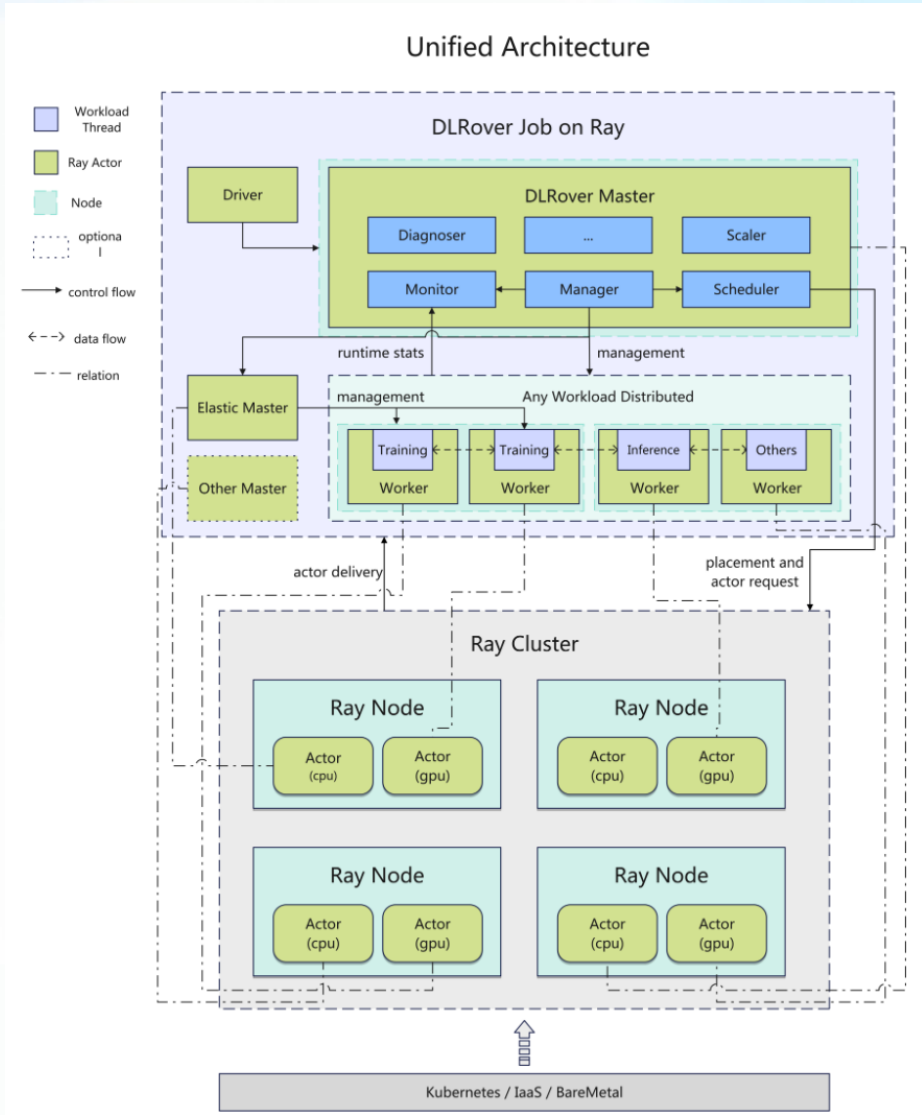
1. 原生分布式能力：基于其原生的分布式能力，框架层可轻松表达异构的进程级负载，也能更轻松的实现诸如容错等上层建设
2. 更灵活且自然的分布式交互表达：基于ray.remote的方式不再需要框架层自行构建server/client的交互实现，大幅简化了异构计算的实现成本
3. 更灵活的调度支持：基于placement group的方式可自由组合负载进程，使异构负载在异构资源环境上的自由组合成为可能
4. 健全的生态能力：借助Ray一方的原生上层能力，如Ray-Data等，能够覆盖更多复杂融合场景下的计算实现
5. Python语音亲和性更高（相比kubernetes的golang）

On Ray的设计与实现

新的基于Ray的架构如右图所示，核心区别：

1. 负载->任意类型
2. 机器->进程

- 异构的控制面和负载
- 算法松耦合
- 更加灵活的调度
- 更加灵活的容错
- 统一的API



On Ray的设计与实现

控制面：一拆多

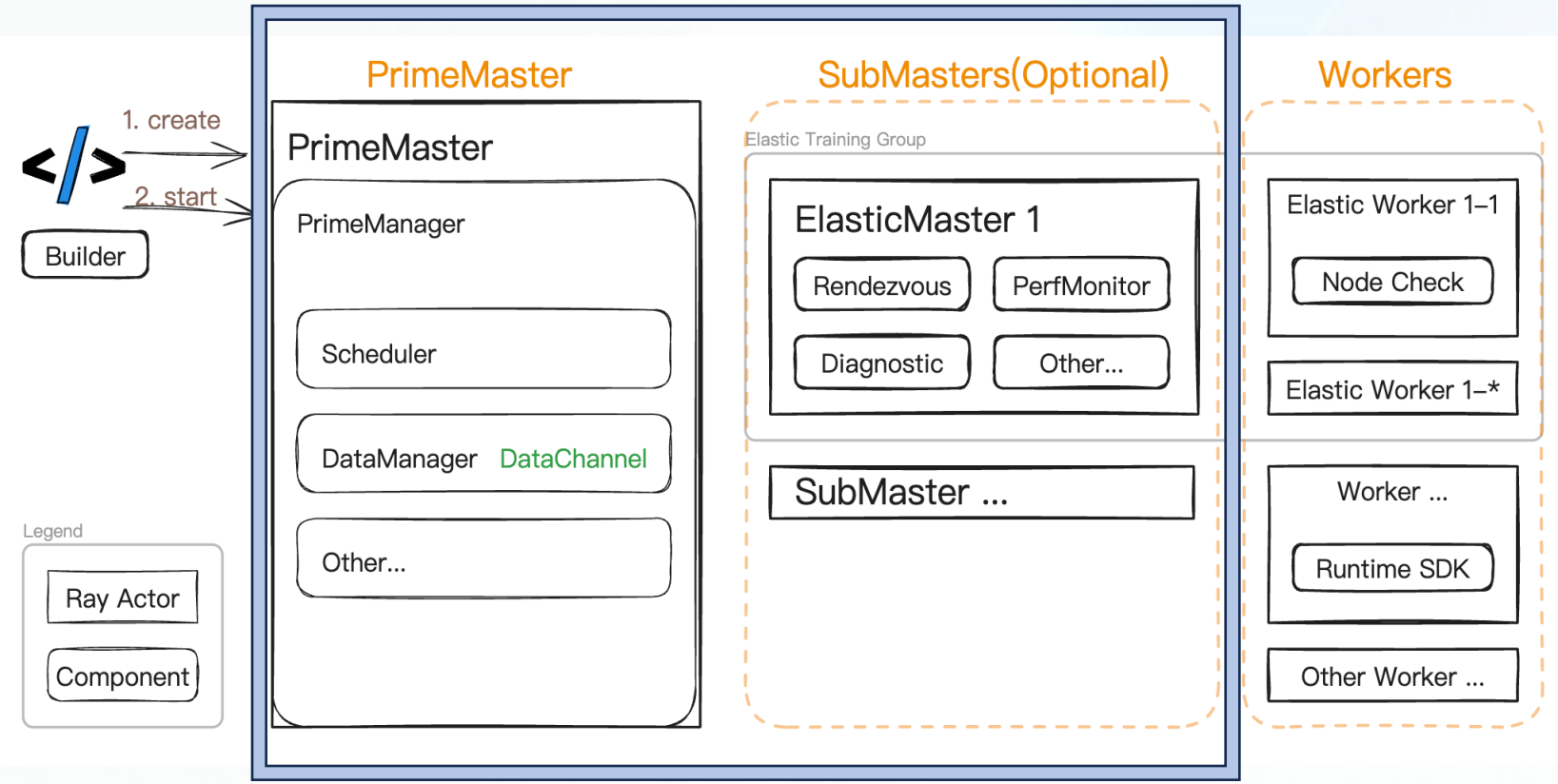
- PrimeMaster：

统一的控制面操作，如Job生命周期管理，Actor调度，Job级容错和诊断等。

- SubMasters(Optional)：

面向具体计算组的控制面扩展。例如：ElasticMaster是针对当前torchrun类型的弹性训练组的控制面扩展。

异构的控制面



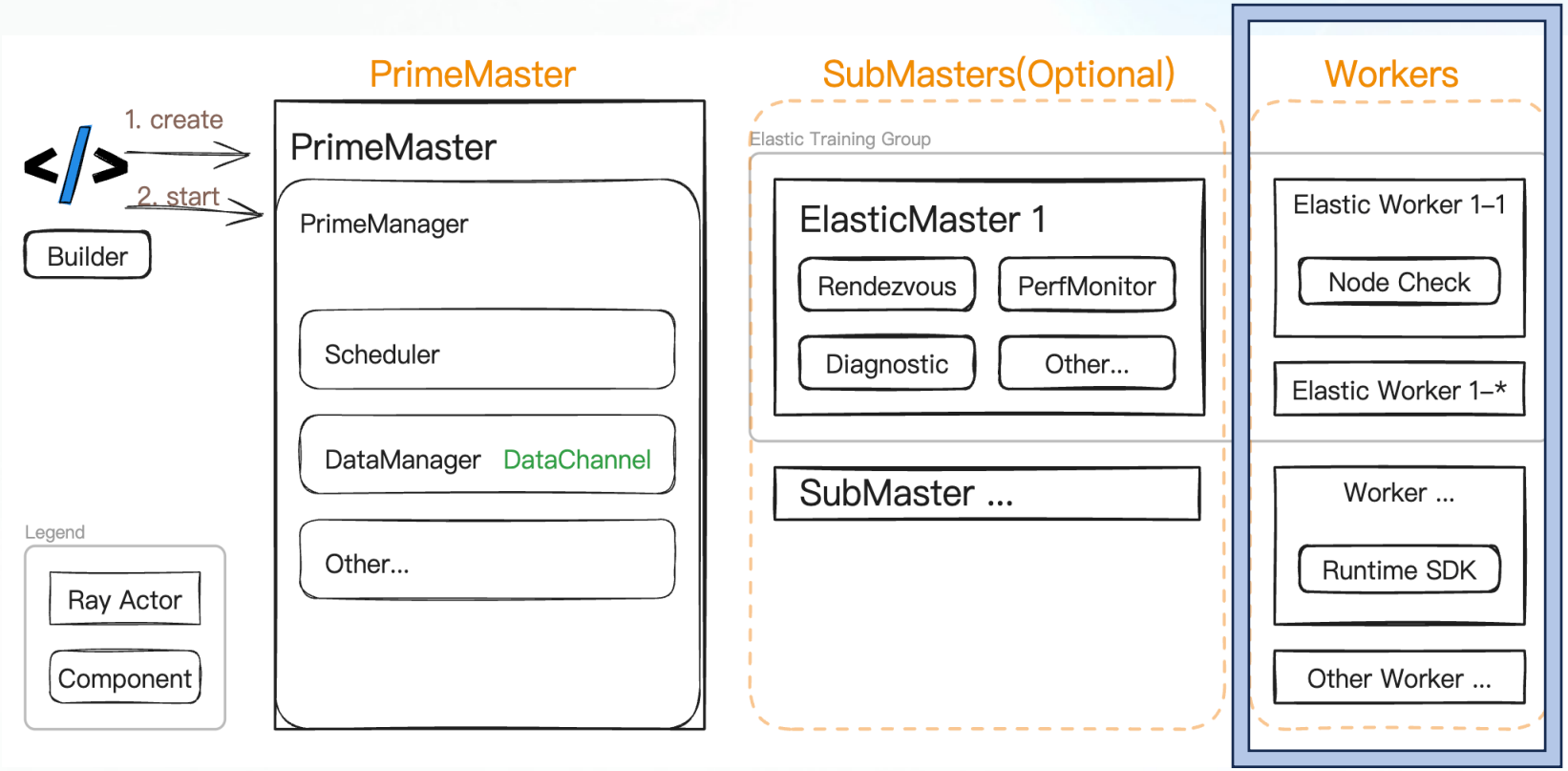
On Ray的设计与实现

异构的负载

负载：一变多
一种 ElasticAgent
->
N种 Roles

Role -> Role Group:
N个Actor进程的集合

Role A: Actor Rank0
remote call
Role B: Actor All/Rank0
with dispatching strategy



On Ray的设计与实现

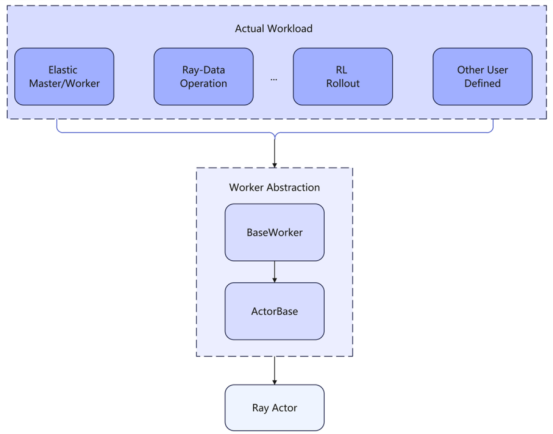
算法松耦合

DLRover: 工程框架 not 算法框架

算法框架：开箱即用

环境准备，配置yaml -> 发起训练

```
ray job submit --address="http://127.0.0.1:8265" \
   --runtime-env-json='{"working_dir": "/openrlhf"}' \
   -- python3 -m openrlhf.cli.train_ppo_ray \
   --ref_num_nodes 1 \
   --ref_num_gpus_per_node 8 \
   --reward_num_nodes 1 \
   --reward_num_gpus_per_node 8 \
   --critic_num_nodes 1 \
   --critic_num_gpus_per_node 8 \
   --actor_num_nodes 1 \
   --actor_num_gpus_per_node 8 \
   --vllm_num_engines 4 \
   --vllm_tensor_parallel_size 2 \
   --colocate_all_models \
   --vllm_gpu_memory_utilization 0.5 \
   --pretrain OpenRLHF/Llama-3-8b-sft-mixture \
   --reward_pretrain OpenRLHF/Llama-3-8b-rm-700k \
   --save_path /openrlhf/examples/test_scripts/final/llama3-8b-rlhf \
   --ckpt_path /openrlhf/examples/test_scripts/ckpt/llama3-8b-rlhf \
   --save_hf_ckpt \
   --micro_train_batch_size 8 \
   --train_batch_size 128 \
```



依赖完整的算法编程，基于DLRover提供的基类 - 以实现不同的workload。DLRover基类可提供一些通用能力：

- 负载的基础信息
- 自检/容错的基础能力
- On Torch计算的基础能力封装

On Ray的设计与实现

蚂蚁开源 ANT OPEN SOURCE

RAY

样例:OpenRLHF-ppo

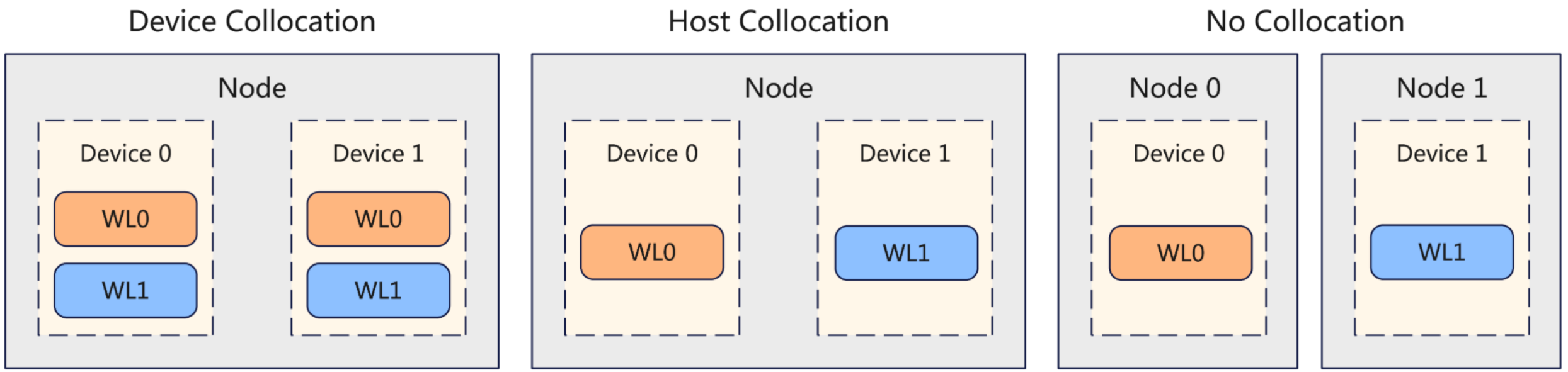
```python
# examples/unified/rl/openrlhf/ppo/main.py
# Inside submit()

# 1. Create a job builder
builder = RLJobBuilder().config(args_2_omega_conf(args))

# 2. Define workloads / roles
builder.trainer(f"{P}.ppo_trainer.PPOTrainerActor").resource(cpu=4)
(
    builder.actor(f"{P}.ppo_actor.PolicyModelActor")
    .resource(accelerator=1)
    .nnodes(args.actor_num_nodes)
    .nproc_per_node(args.actor_num_gpus_per_node)
)
(
    builder.critic(f"{P}.ppo_critic.CriticModelRayActor")
    .resource(accelerator=0.4)
    .nnodes(args.critic_num_nodes)
    .nproc_per_node(args.critic_num_gpus_per_node)
)
(
    builder.rollout(f"{P}.ppo_rollout.VLLMActor")
    .resource(accelerator=1)
    .total(args.vllm_num_engines)
)
# optional reference role
if args.init_kl_coef != 0:
    (
        builder.role(RLRoleType.REFERENCE.name)
        .train("{P}.ppo_reference.PPOReferenceActor")
        .resource(accelerator=0.4)
        .nnodes(args.ref_num_nodes)
        .nproc_per_node(args.ref_num_gpus_per_node)
    )
(
    builder.role(RLRoleType.REWARD.name)
    .train("{P}.ppo_reward.RewardModelRayActor")
    .resource(accelerator=0.4)
    .nnodes(args.reward_num_nodes)
    .nproc_per_node(args.reward_num_gpus_per_node)
)

# 3. Build and submit
job = builder.build()
# optional modifications before submission
if args.skip_node_check:
    for workload in job.workloads.values():
        if workload.backend == "elastic":
            workload.comm_pre_check = False
print(job.model_dump_json(indent=2))
if not args.dry_run:
  job.submit(args.job_name)
```

On Ray的设计与实现

更加灵活的调度

面向异构场景，基于Ray的Placement Group的能力，支持了常见的亲和和反亲和调度：

1. 设备内的亲和调度 2. 同机设备间的亲和调度 3. 机间的反亲和调度



On Ray的设计与实现

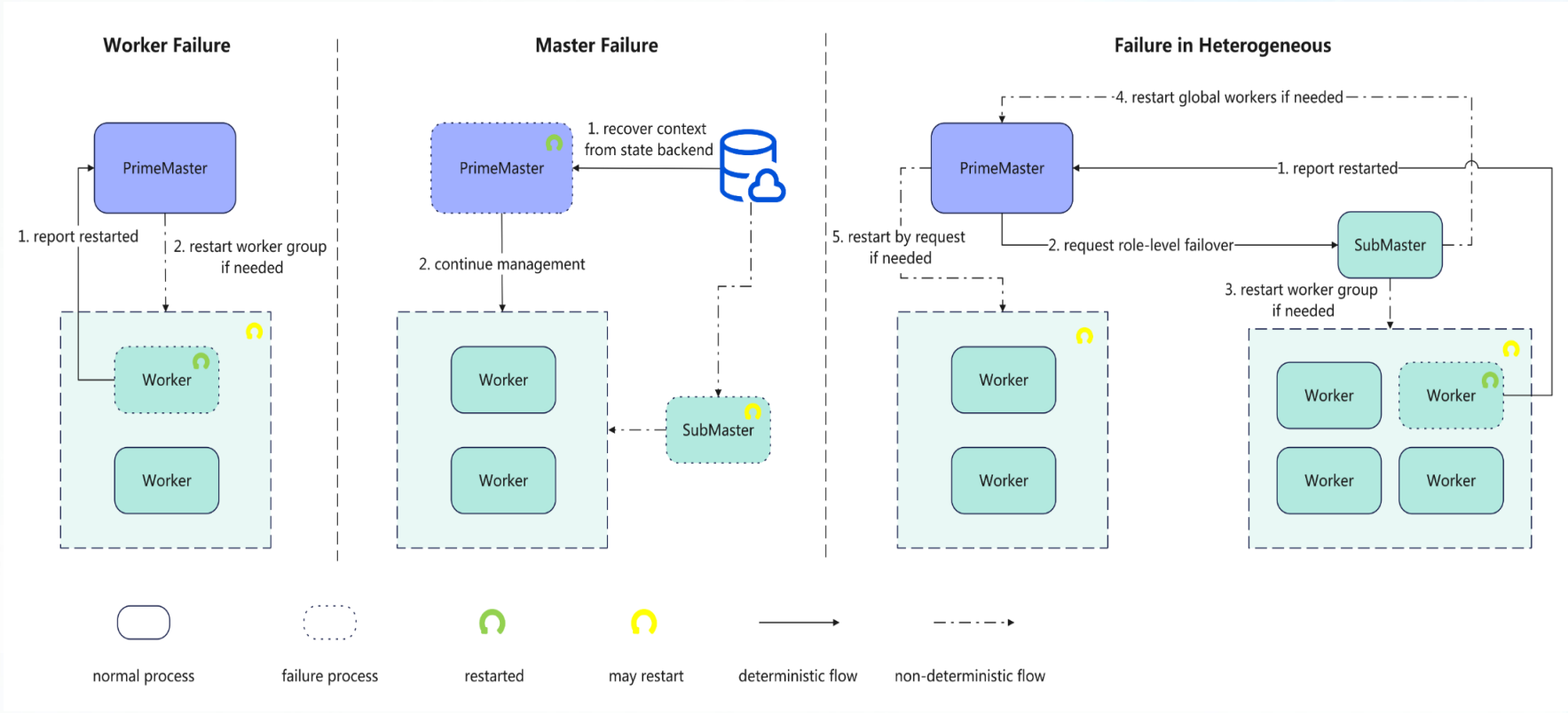
更加灵活的容错

控制面：

- 原生Master容错
  （依赖可持久化的运行时上下文）

负载：

- Job级容错
- Group级容错
- 单点容错（？）



On Ray的设计与实现

1. 充分灵活的表达
2. 减少用户面向不同场景的使用成本

- 任意类型计算负载的表达
- 可扩展的控制面实现
- 多种调度策略
- 进程级运行时环境变量
- 进程级资源管理
- 进程级参数配置
- 存量torchrun使用方式低成本迁移

统一的API

```
from dlrover.python.unified.api.builder.base import DLJobBuilder


dl_job = (
    DLJobBuilder()
    .node_num(worker_node_num) # 整体的节点数量
    .device_per_node(device_per_worker_node)  # 单位节点的设备数量
    .device_type("GPU") # 设备类型
    .config({}) # 用户自定义的全局配置
    .global_env({"DLROVER_LOG_LEVEL": log_level}) # 用户自定义的全局环境变量
    .role(xxx).run(xxx) # 任意负载
      .resource(xxx) # 负载的自定义资源
      .total(xxx) # 负载的总数
      .per_group(xxx) # 负载的组内数量
    .role(xxx).run(xxx) # 任意负载
      ...
    .workload("role","entrypoint") # 任意负载（换个形式表达）
      ...
    ...
    .build()
)
```

未来计划

- 年底发布第一个基于Ray架构实现的正式版本

- 完善现有设计与实现
  - 强化现有能力，包括：调度、容错等。
  - 增加异构场景下的运行时诊断实现，包括和XPU_TIMER的联动等能力。

- 子项目扩展：针对部分算法或场景扩展独立的子开源项目，与DLRover共建，例如：强化学习和多模态场景

- 离在线场景的统一：结合现有基于Ray链路“在线学习”的实践经验，在DLRover中融合离在线两种场景的实现。

欢迎大家一起交流

DLRover社区开源交流群

个人联系方式