TYDocs

fixer

2023 年 01 月 16 日

目录:

CHAPTER 1

功能

1.1 简介

TY(TAIYI)-Arranger 是基于 kubebuilder 编写的容器批量计算控制器，用于分布式强化学习训练框架在集群上的任务部署。目前的分布式强化学习训练框架主要专注于算法的集成，在计算资源尤其是多机资源的协调利用方面支持欠佳。TY-Arranger 为 TYRL(TAIYI Reinforcement Learning Framework) 在复杂任务训练场景上的应用提供支持，为其提供 CPU,GPU 等资源的调度能力。

1.2 TYRL

1.3 特性

支持多种 pod 调度策略，包括

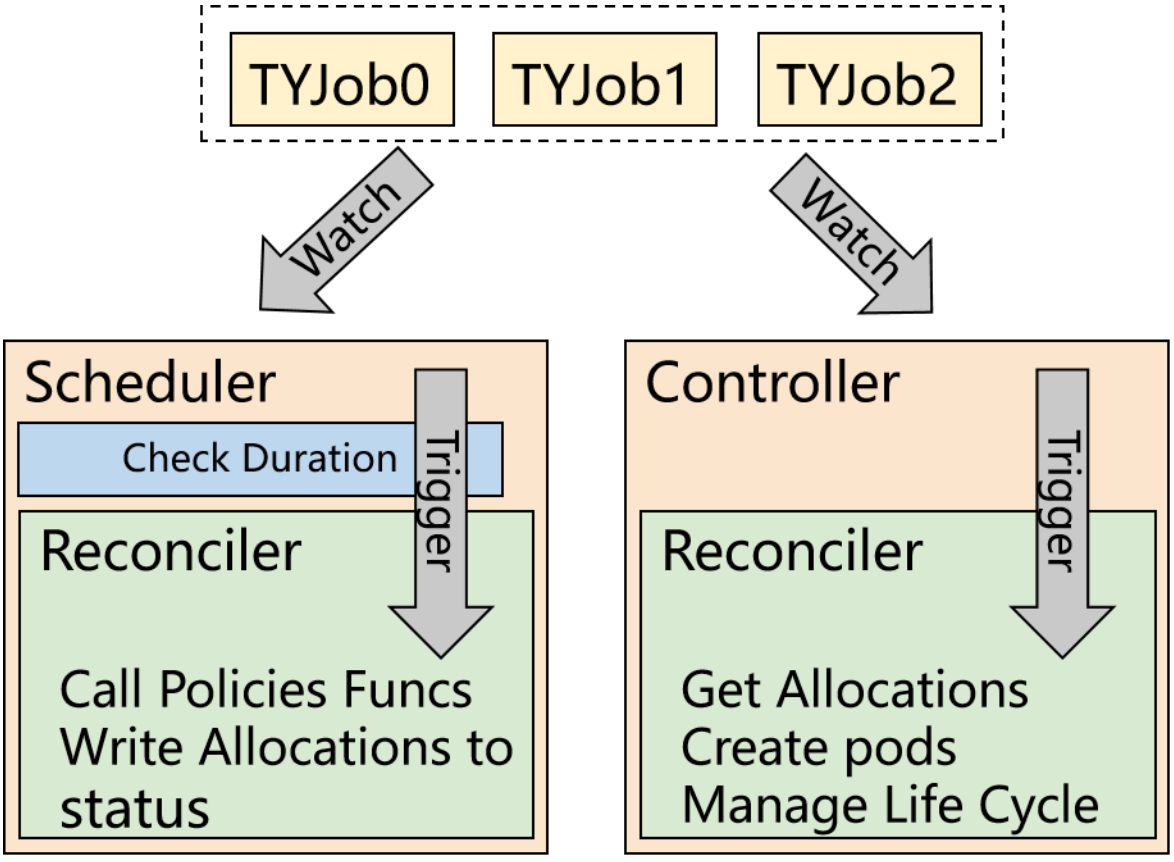
- Binpack policy
- LeaderFirst policy
- MinFragmen policy
- JobAffinity Policy
- JobAntiAffinity

支持多种 Job 调度策略，包括

- PriorityPolicy
- DRFPolicy

CHAPTER 2

架构



Arranger 由 Scheduler 和 Controller 两部分组成：

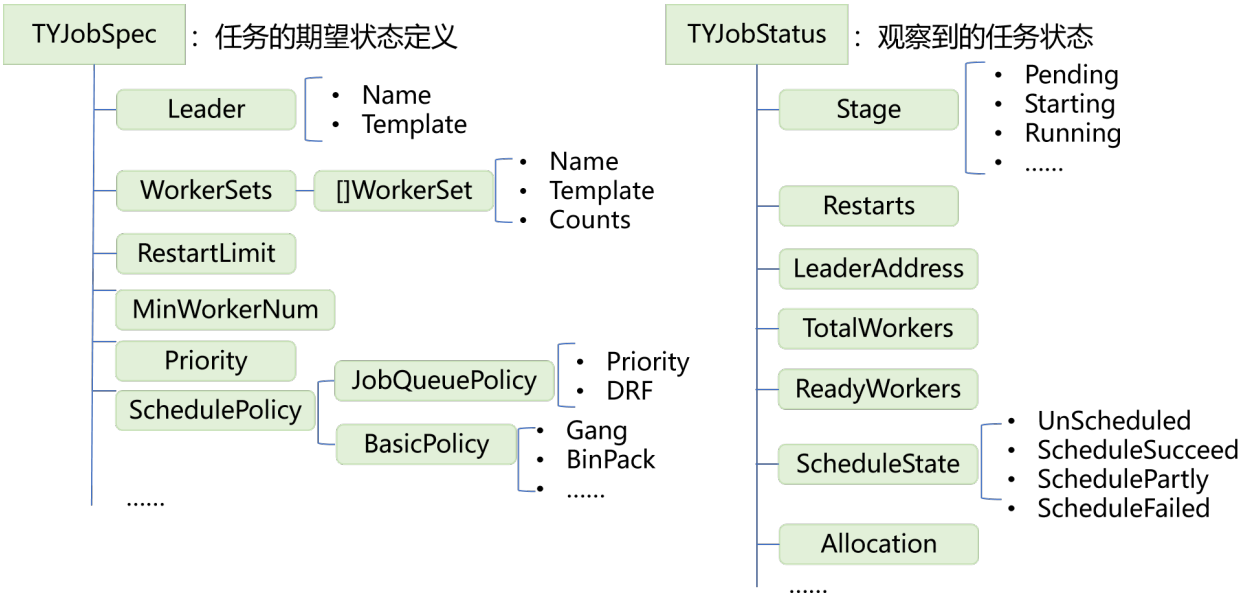
- Controller 负责对 CRD API 资源进行合法性校验，管理 CRD（TYJob）的生命周期，创建和删除资

源。

- Scheduler 是一个周期性调度器，会将周期内的 job 依策略排序，按顺序对其中的任务进行调度，并支持多种调度算法。

2.1 API

Arranger 针对 TYRL 的结构设计了自定义资源 TYJob，Leader 对应 TYRL 中的 Learner，Worker 则对应 Actor，WorkerSet 是一类启动参数的 Actor 的集合。.. image:: images/TYJob 结构.png



TYJobSpec 定义了任务的期望状态，各字段及其含义如下：

```
type TYJobSpec struct {
    // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
    // Important: Run "make" to regenerate code after modifying this file

    // RestartLimit defines the restart limit for Leader and WorkerSet
    // +kubebuilder:default=3
    RestartLimit int `json:"restartLimit,omitempty"`

    // Volumes defines the shared volumes for job.
    Volumes []corev1.Volume `json:"volumes,omitempty"`

    // leader defines learner and manger pod
    // +kubebuilder:validation:Required
    Leader Leader `json:"leader,omitempty"`

    //worker defines actor set
```

(续下页)

(接上页)

```
    // +kubebuilder:validation:Required
    WorkerSets []WorkerSet `json:"workerSets,omitempty"`

    // CleanPodPolicy defines the policy to clean pods after TYJob completed.
    // +kubebuilder:default=All
    // +kubebuilder:validation:Enum=Running;All;None
    CleanPodPolicy CleanPodPolicy `json:"cleanPodPolicy,omitempty"`

    // handle terminate.
    //+kubebuilder:default=false
    Terminating bool `json:"terminating,omitempty"`

    // MinWorkersNum defines the minimum num of workers to start the job
    //+kubebuilder:default=1
    MinWorkersNum int `json:"minWorkersNum,omitempty"`

    //Priority defines the priority of the job
    //+kubebuilder:default=5
    //+kubebuilder:validation:Minimum=1
    //+kubebuilder:validation:Maximum=10
    Priority int `json:"priority,omitempty"`

    SchedulerPolicy Policy `json:"schedulerPolicy,omitempty"`
}
type Policy struct {
    //JobQueuePolicy defines the schedule policy among jobs
    JobQueuePolicy JobQueuePolicy `json:"jobQueuePolicy,omitempty"`

    //BasicPolicy defines the schedule policy of one job
    BasicPolicy BasicPolicy `json:"basicPolicy,omitempty"`
}
type JobQueuePolicy string

const (
    PriorityPolicy JobQueuePolicy = "Priority"

    DRFPolicy JobQueuePolicy = "DRF"
)

type BasicPolicy string

const (
    GangPolicy BasicPolicy = "Gang"
```

(续下页)

(接上页)

```
    BinPackPolicy BasicPolicy = "BinPack"

    LeaderFirstPolicy BasicPolicy = "LeaderFirst"

    MinFragmentPolicy BasicPolicy = "MinFragment"

    JobAffinityPolicy BasicPolicy = "JobAffinity"

    JobAntiAffinity BasicPolicy = "JobAntiAffinity"
)

type CleanPodPolicy string

const (
    // CleanPodPolicyRunning means deleting all running pods of the job after completed
    CleanPodPolicyRunning CleanPodPolicy = "Running"

    // CleanPodPolicyAll means deleting all pods of the job after completed
    CleanPodPolicyAll CleanPodPolicy = "All"

    // CleanPodPolicyNone means never deleting any pods of the job after completed
    CleanPodPolicyNone CleanPodPolicy = "None"
)

type Leader struct {
    // +kubebuilder:validation:Required
    Name string `json:"name,omitempty"`

    // Template defines the learner pod for TYJob.
    // +kubebuilder:validation:Required
    Template corev1.PodTemplateSpec `json:"template,omitempty"`
}

type WorkerSet struct {
    // +kubebuilder:validation:Required
    Name string `json:"name,omitempty"`

    // Template defines the actor pod for TYJob.
    // +kubebuilder:validation:Required
    Template corev1.PodTemplateSpec `json:"template,omitempty"`

    // Counts defines the number of workers in workSet.
```

(续下页)

(接上页)

```
    // +kubebuilder:default=1
    // +kubebuilder:validation:Minimum=1
    Counts int `json:"counts,omitempty"`
}
```

Controller 控制 TYJob 的生命周期，TYJob 的状态 Stage 设计如下：

```
type Stage string

const (
    // JobPending means the job has been submitted to the cluster,
    JobPending Stage = "Pending"

    // JobStarting means the leader has been created and waits for creating␣
↪workers(writing env -> leader ip)
    JobStarting Stage = "Starting"

    // JobRunning means all the pods are in running state
    JobRunning Stage = "Running"

    // JobRescheduling means the job has been rescheduled and waits for restarting.
    JobRescheduling Stage = "Rescheduling"

    // JobSucceeded means job completed without error
    JobSucceeded Stage = "Succeeded"

    // JobFailed means some pods failed, job is also considered failed
    JobFailed Stage = "Failed"
)
```

Scheduler 控制 TYJob 的调度阶段，调度阶段设计如下：

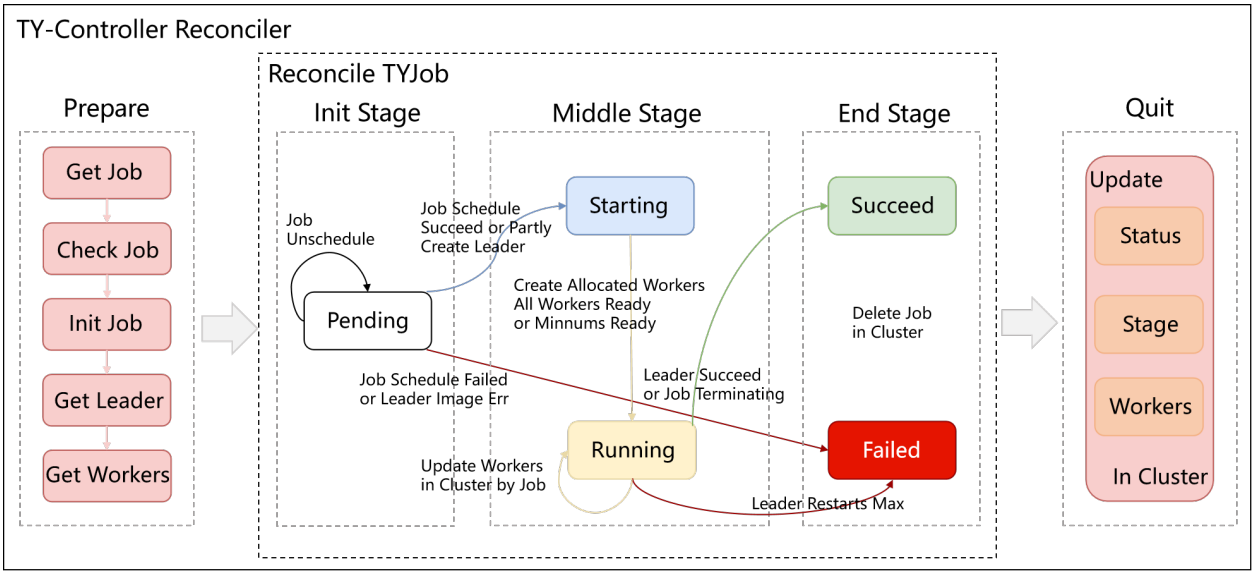
```
type ScheduleState string

const (
    JobUnscheduled ScheduleState = "Unscheduled"

    JobScheduledSucceed ScheduleState = "ScheduledSucceed"

    JobScheduledFailed ScheduleState = "ScheduledFailed"

    JobScheduledPartly ScheduleState = "ScheduledPartly"
)
```

2.2 Controller

2.2.1 简介

Arranger Controller 负责管理 Job 的生命周期即 Stage，包括 Pending,Starting,Running,Succeed,Failed，同时负责状态中 Leader 和 Worker 的创建及删除。在一次 Reconciler 中，Controller 获取触发 Reconciler 的 Job 对象，检查声明资源的合法性并进行初始化，继而根据当前的 Stage 和调度阶段进行状态转移，最后更新 Status,Stage,Workers，结束此次 Reconciler

2.2.2 工作流



Arranger Controller 工作流程如下：

1. 获取用户提交的 Job，检查合法性并初始化
2. 状态转移
    - TYJob 初次提交时处于 Pending 阶段
    - 若此时的调度阶段处于 UnScheduled，保持 Pending 状态
    - 若此时的调度阶段处于 ScheduledPartly 或 ScheduledScuueed 阶段，创建 Leader，Job 进入 Starting 状态
    - 若此时的调度阶段处于 ScheduledFailed 阶段，进入 Failed 状态
    - Starting 阶段会创建所有已有调度结果的 Worker，当所有或最小需要的 Worker 都处于 Ready 状态时 Job 进入 Running 状态
    - Running 阶段会尝试创建未创建的 Worker（若存在）
    - 当 Leader Succeed 或 Job Terminating 时，Job 进入 Succeed 状态

- 当 Leader 重启次数超过最大重启次数时，Job 进入 Failed 状态
- Job 到达 Succeed 或 Failed 状态后会清空所有 Pod 资源，任务结束

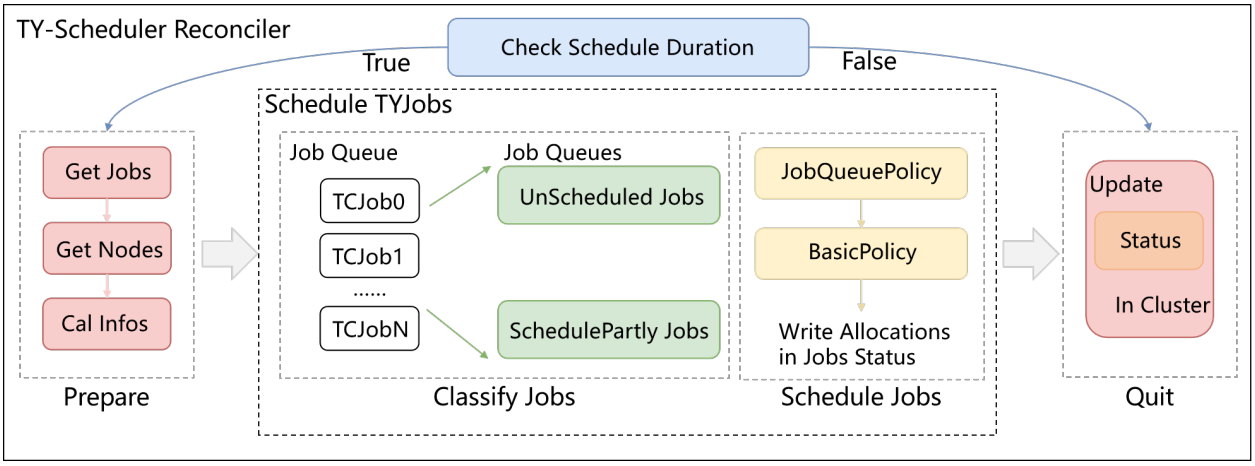
3. 更新 Status,Stage,Workers，结束本次 Reconciler

2.3 Scheduler

2.3.1 简介

Arranger Scheduler 负责调度 job 和 pod，两种调度均支持多种调度策略。在一次 Reconciler 中，Scheduler 先判断是否到达调度周期。若到达，则准备调度需要的 Jobs 及 Nodes 信息。Job Queue 会依 Job 调度策略对 Job 进行排序，再依次处理 Job 中的 Pod。Job 根据其中 Pods 的调度阶段确定本身的调度阶段，包括 Unscheduled,ScheduledSucceed,ScheduleFailed,ScheduledPartly, Scheduler 会尝试为处于后两个调度阶段的 job 中的未调度 pod 进行调度，并将结果写入 TYJob.status.Allocations 中以供 Controller 使用。

2.3.2 工作流



Arranger Scheduler 工作流程如下：

1. 周期性地开启调度
2. 获取用户提交的 Job 和节点信息
3. 将 Job 按调度阶段分类
4. 根据 JobQueuePolicy 对 Job Queue 进行排序，再根据 BasicPolicy 对 Job 中的 Pod 进行调度，找到当前策略下得分最高的节点，保存该结果
5. 更新 Status，结束本次 Reconcile

2.3.3 JobQueuePolicy

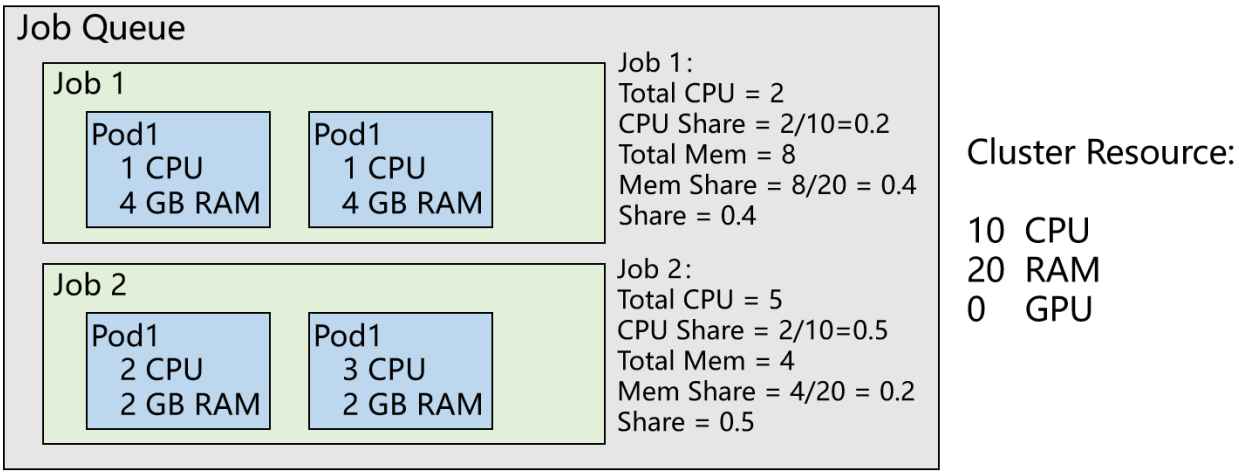
Arranger 支持在一个调度周期中对多 job 任务进行调度，这些 job 会根据其中任务的调度情况打上调度阶段标签，所有待调度 Job 将根据 JobQueuePolicy 进行排序，以下为目前支持的策略：

PriorityPolicy

根据 JobQueue 中每个 Job.Spec.Priority 的值对 Queue 中所有 Job 排序，确定被 Scheduler 调度的先后顺序

DRFPolicy

Dominant Resource Fairness，Scheduler 根据 Job 的主导资源，计算 Job 的 share 值，在调度的过程中，具有较低 share 值的 Job 将具有更高的调度优先级



2.3.4 BasicPolicy

在对单个 pod 进行调度时，Arranger 对所有能够满足需要资源的节点按一定标准进行打分，将 pod 调度到得分最高的节点上，以下为目前支持的策略：

Binpack policy

目标是尽量把已有的节点填满（尽量不往空白节点分配）。具体实现上，Binpack 调度算法是给可以投递的 Nodes 打分，分数越高表示节点的资源利用率越高，将应用负载靠拢在部分 Nodes

LeaderFirst policy

目标是将 Learner 调度到资源利用率较低的 Node，保障 Learner 的正常运行；Actor 则调度到资源利用率较大的 Node，节约部分资源。Leader GPU 权重提高，选分数低的节点，Worker CPU 权重提高，选分数高的节点

MinFragmen policy

目标是减少集群节点的资源碎片率。碎片率：{ 1 - Abs[CPU(Request / Allocatable) - Mem(Request / Allocatable)] } * Score。是用来考虑 CPU 的使用比例和内存使用比例的差值。如果这个差值越大，就表示碎片越大，Worker 优先不分配到这个节点上；如果这个差值越小，就表示这个碎片率越小，Worker 优先分配到这个节点上

JobAffinity Policy

目标是尽量将一个 Job 的 Pods 调度到一个节点，集中负载

JobAntiAffinity

目标是尽量将一个 Job 的 Pods 打散到不同节点，平坦负载

CHAPTER 3

开始

3.1 安装

镜像拉取，operator 安装

3.2 快速开始

3.2.1 创建任务

3.2.2 检查状态

3.2.3 删除任务