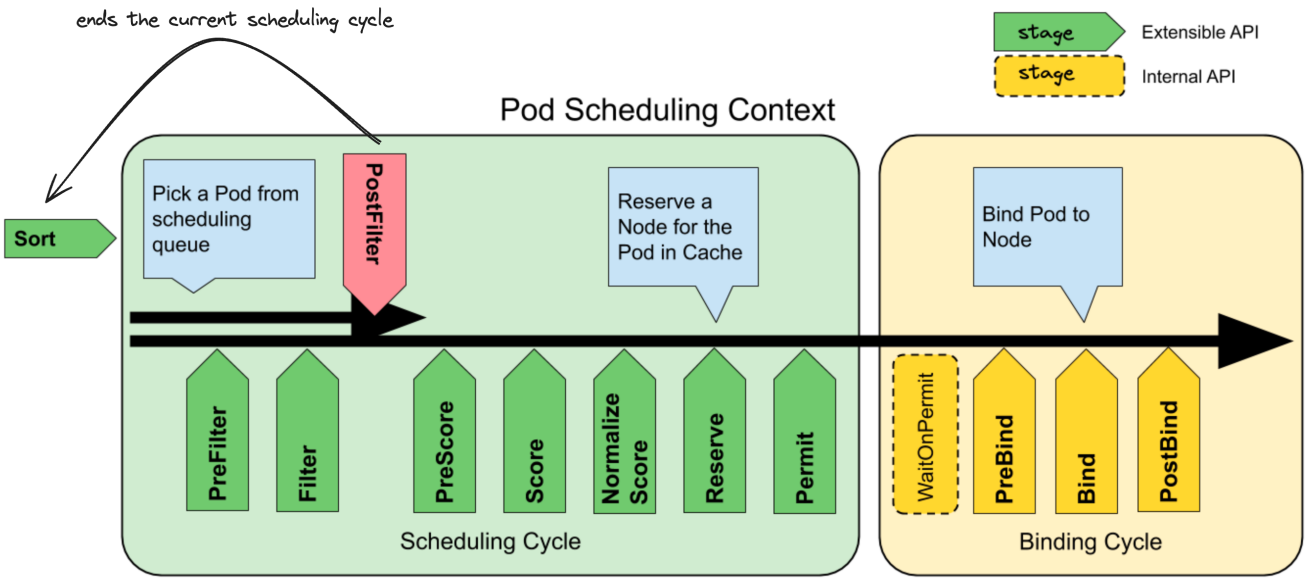
k8s scheduler

简介



阶段

- PreEnqueue: 在对pod调度前，检查pod并决定是否放入队列，如新的drm功能，在pod.spec 中有新的定义 resourceClaim，pod要求必须有资源才会调度。
  - 当前有 dynamicresources 和 schedulinggates
  - PreEnqueue(context, v1.Pod) Status
- EnqueueExtensions：集群发生某些事件后，会导致该插件可能失败
  - 当前基本都实现该功能
  - EventsToRegister() []ClusterEventWithHint
- QueueSortPlugin：对调度队列中的pod排序，目前只能使能一个
  - 当前只有queuesort实现
  - Less(QueuedPodInfo, QueuedPodInfo) bool
- preFilterPlugin：调度周期一开始执行，过滤可能节点
  - PreFilter(context, CycleState, Pod) (PreFilterResult, Status)
  - PreFilterExtensions() PreFilterExtensions // Prefilterextension接口
    - AddPod(context, CycleState, podToSchedule Pod, podInfoToAdd PodInfo, nodeInfo NodeInfo) Status

        - RemovePod(context, CycleState, podToSchedule Pod, podInfoToRemove PodInfo, nodeInfo NodeInfo) *Status
- filterPlugin：从可能节点上过滤真正能调读的节点
    - Filter(context, CycleState, pod Pod, nodeInfo NodeInfo) *Status
- postFilterPlugin: 仅当没有为 pod 找到可行节点时调用。如执行过程中将节点标记为可调度，则其余插件将不会被调用。典型实现是抢占，通过抢占其他来使 pod 可调度
    - 当前实现有 dynamicresources，defaultpreemption
- preScorePlugin: 在filter阶段之后，这里也有拒绝可能，当返回skip时，跳过之后的打分过程，进入绑定阶段
    - 当前主要有实现 affinity ，noderesources，podtopologyspread，taint
    - PreScore(context, CycleState, Pod, nodes []v1.Node) Status
- scorePlugin: 对每个节点调用打分，必须返回成功和一个整数，或者拒接
    - Score(context, CycleState, v1.Pod, nodeName string) (int64, Status)
    - ScoreExtensions() ScoreExtensions
- ScoreExtensions：将分数标准化，可以设置插件权重
- ReservePlugin: 发生在实际将 Pod 绑定到指定节点之前，这是为防止调度程序等待绑定成功时出现竞争条件
    - 当前实现有 volumebinding 和 dynamicresources
    - Reserve(context, CycleState, v1.Pod, nodeName string) Status
    - Unreserve(context, CycleState, v1.Pod, nodeName string)
- PermitPlugin：用于阻止或延迟 Pod 的绑定，在filter阶段最后会执行，在bind阶段一开始会执行，为什么会执行两次，因为 filter 和 bind 是独立协程处理
    - 当前未有该实现
    - Permit(context, CycleState, v1.Pod, nodeName string) (Status, time.Duration)
- PreBindPlugin: 在真正执行bind前，如不返回成功则拒绝绑定，这和permit比较类似
    - 当前实现有 volumebinding
    - PreBind(context, CycleState, v1.Pod, nodeName string) Status
- PostBindPlugin：执行完bind后，调用的插件, 主要用于通知
    - 当前实现有 dynamicresources
    - PostBind(context, CycleState, v1.Pod, nodeName string)
- 上面接口中会使用一些通用的数据，其类型介绍如下
    - CycleState：含有跳过的 Filter/Score 插件列表，以及 Storage(主要用于插件不同阶段时交换信息)

- nodeinfo：节点信息(v1.Node)，pod信息，资源(计算后)，计划可以加入更多信息，如存储, 网卡, numa等
- Status： 成功，失败(描述)，不可调度，等待(Permit插件返回类型，调度应等待)，跳过（prefilter，prescore阶段返回时，关联的操作跳过，bind返回时则掉过绑定）

```
// NodeInfo 主要内容
node *v1.Node 描述: 这是一个指向 v1.Node 类型的指针，表示节点的整体信息

Pods []*PodInfo 描述: 这是一个 PodInfo 类型的切片，表示在节点上运行的所有 Pod。

PodsWithAffinity []*PodInfo 描述: 这是一个 PodInfo 类型的切片，表示在节点上运行的具有亲和性（Affinity）的 Pod 子集。

PodsWithRequiredAntiAffinity []*PodInfo 这是一个 PodInfo 类型的切片，表示在节点上运行的具有强制反亲和性（Required Anti-Affinity）的 Pod 子集。

Requested *Resource:
描述: 这个字段表示节点上所有 Pod 的资源请求总量。
用途: 用于记录节点上所有 Pod 的资源请求总量，以便调度器进行资源分配和调度决策。

NonZeroRequested *Resource:
描述: 这个字段表示节点上所有 Pod 的资源请求总量，但每个容器的 CPU 和内存请求都有一个最小值，即未设置 request 的 pod
用途: 这个字段并不反映节点上实际的资源请求，而是用于避免将许多零请求的 Pod 调度到同一个节点上
```

逻辑

- framework 目录下提供能力，包括plugin(注册，初始化时需要的handler，args校验)，各阶段的入口调用，调用逻辑在 pkg/scheduler 目录下的 scheduler_one 文件中
- podQueue 使用三种队列，具体设计文档，主要是排除需要调度的pod
  - PreEnqueue 当pod要求的资源不满足时不入调度队列
  - EnqueueExtensions 当环境中发生某些事件时，pod入调度队列
  - QueueSortPlugin
- 执行调度循环前，会有以下动作
  - 执行filter操作前会选择部分主机，在 100 数量以下的直接返回数量，默认返回50%
  - 做节点快照，提高效率，降低延迟
- findNodesThatFitPod：
  - 依次执行 prefilter 排除不可能节点，这里可能会有潜在单节点，优先执行 filter 和extender 并返回
  - 这里会挑选100或50%以内主机 parallelize 并行 filter 过滤可调度节点

    - 依次 extender 并发送 pod+nodes
- prioritizeNodes：对可调度节点打分
    - 依次执行 preScore 、Score 、Extender 进行打分，通常 preScore 即打分也过滤，Score 使用 preScore 中的信息
- 详细介绍
- nodeResource Score 插件
    - 三种策略 LeastAllocated(默认)，MostAllocated 和 RequestedToCapacityRatio, 并且对于 cpu，memory 资源，权重默认 1
    - 以下计算公式中 capacity: 节点可分配，requested: 节点已分配+pod请求
    - leastResourceScorer 目标是选择剩余资源较多节点，公式如下
        - (cpu((capacity-requested) * MaxNodeScore * cpuWeight / capacity) + memory((capacity-requested) * MaxNodeScore * memoryWeight / capacity) + ...)/weightSum
    - mostResourceScorer 目标是选择已经被分配了较多资源的节点。这种方法有助于确保资源密集型任务能够被调度到资源充足的节点上，从而提高集群的整体资源利用率，公式如下
        - (cpu(MaxNodeScore * requested * cpuWeight / capacity) + memory(MaxNodeScore * requested * memoryWeight / capacity) + ...) / weightSum
- balanceResource Score 插件
    - fraction 分配比例，某类资源的 requested / capacity
    - mean 平均值，Σ(fraction)/len(fractions)
    - 当两种即以下资源类型，公式
        - (fraction1-fraction2)/2
    - 大于两种资源类型时，公式
        - sqrt(Σ((fraction(i)-mean)^2)/len(fractions))
- 标准差: 统计学中用于衡量数据分布的离散程度的一种度量, 反映数据点相对于平均值的分散程度, 标准差越大，数据点越分散；标准差越小，数据点越集中。主要用于评估资源分配的平衡性，标准差越小，表示资源分配越平衡

配置

- https://kubernetes.io/docs/reference/scheduling/config/
    - 介绍：配置文件格式; 扩展点支持的插件位置; 多配置策略
- 当某个插件支持 preScore, score, preFilter, 和 filter 时，可以放入 multiPoint 下
- kubescheduler.config.k8s.io 可配置内容
    - parallelism: 默认 16
    - leaderElection: 配置选举策略
    - clientConnection 客户端连接方式

    - percentageOfNodesToScore：当总节点数这个百分比ready后既可参与调度，否则等待 ready 后开始
    - podInitialBackoffSeconds: 退避初始时间，默认 1秒
    - podMaxBackoffSeconds：最大退避时间，默认 10秒
    - profiles：列表格式，定义启用/禁用的插件，插件配置
    - extenders：定义外部插件，verb 后缀是请求端点
        - urlPrefix
        - filterVerb
        - preemptVerb
        - prioritizeVerb
        - bindVerb
        - enableHTTPS
        - httpTimeout
        - ignorable：当失败或连接不成功时，是否忽略该插件
        - weight
    - delayCacheUntilActive：默认false，当成功选主后，再开始缓存信息
- 默认使用的插件，自 1.25 版本稳定
    - ImageLocality：优先考虑已经拥有 Pod 运行的容器镜像的节点。 延伸点：分数。
    - TaintToleration：实现污点和容忍。 实现扩展点：filter、preScore、score。
    - NodeName：检查 Pod 规范节点名称是否与当前节点匹配。 延伸点：过滤器。
    - NodePorts：检查节点是否有空闲端口用于请求的 Pod 端口。 延伸点：预过滤器、过滤器。
    - NodeAffinity：实现节点选择器和节点亲和性。 扩展点：过滤、评分。
    - PodTopologySpread：实现Pod拓扑传播。 扩展点：preFilter、filter、preScore、score。
    - NodeUnschedulable：过滤掉 .spec.unschedulable 设置为 true 的节点。 延伸点：过滤器。
    - NodeResourcesFit：检查节点是否拥有 Pod 请求的所有资源。扩展点：preFilter、filter、score。
    - NodeResourcesBalancedAllocation：如果将 Pod 调度到那里，则有利于那些可以获得更平衡资源使用的节点。 延伸点：分数。
    - VolumeBinding：检查节点是否具有或是否可以绑定请求的卷。 扩展点：preFilter、filter、reserve、preBind、score。
    - VolumeRestrictions：检查节点中安装的卷是否满足特定于卷提供程序的限制。 延伸点：过滤器。
    - VolumeZone：检查请求的卷是否满足它们可能具有的任何区域要求。 延伸点：过滤器。
    - NodeVolumeLimits：检查节点是否可以满足 CSI 卷限制。 延伸点：过滤器。
    - EBSLimits：检查节点是否可以满足 AWS EBS 卷限制。 延伸点：过滤器。

- GCEPDLimits：检查节点是否可以满足 GCP-PD 卷限制。 延伸点：过滤器。
- AzureDiskLimits：检查节点是否可以满足 Azure 磁盘卷限制。 延伸点：过滤器。
- InterPodAffinity：实现 Pod 间亲和性和反亲和性。 扩展点：preFilter、filter、preScore、score。
- PrioritySort：提供默认的基于优先级的排序。 扩展点：queueSort。
- DefaultBinder：提供默认的绑定机制。 扩展点：绑定。
- DefaultPreemption：提供默认的抢占机制。 扩展点：postFilter。

```
--config: 指定配置文件，文件格式 kubescheduler.config.k8s.io
--scheduler-name: 指定调度器名称，默认 default-scheduler

// 配置文件例子
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
  percentageOfNodesToScore:
  - pluginConfig:
    - args:
        scoringStrategy:
          resources:
          - name: cpu
            weight: 1
          type: MostAllocated
      name: NodeResourcesFit
  - schedulerName: default-scheduler
  - schedulerName: no-scoring-scheduler
    plugins:
      multiPoint:
        enabled:
        - name: MyPlugin
      preScore:
        disabled:
        - name: '*'
      score:
        disabled:
        - name: '*'
        enabled:
        - name: MyCustomPlugin
```

观测

- 使用

    - https://kwok.sigs.k8s.io/
    - https://github.com/kubernetes-sigs/kube-scheduler-simulator 模拟执行结果
- 通过 10259 端口获取指标，常用指标如下
- scheduling_algorithm_duration_seconds： 执行filter/score的耗时
- scheduler_plugin_execution_duration_seconds：每个扩展点执行耗时
- workqueue_queue_duration_seconds: 工作队列耗时

扩展

webhook

- 通过配置 extender 实现

wasm

- https://github.com/kubernetes-sigs/kube-scheduler-wasm-extension
- 使用 framework 每次需要重新编译，通过wasm方式，不再需要重新编译

frame

- 实现接口，并需编译kube-scheduler，参考文档
    - https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/
    - https://github.com/kubernetes-sigs/scheduler-plugins
- 在 scheduler-plugins 中实现较多新的插件，但其阶段和框架不变
- capacity：https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/kep/9-capacity-scheduling
    - 背景：ResourceQuota 中根据资源配置（request.cpu/mem ）进行限制，虽然可以保证实际资源消耗永远不会超过ResourceQuota限制，但可能会导致资源利用率较低，因为某些Pod可能已声明资源但无法调度，为了克服上述限制，将 Yarn 容量调度程序中使用的“ElasticQuota”概念来使用，引入 min/max 其中 min <= max <= resourceQuota，二者主要是保证从其他quota可以借用的范围
    - prefilter: 检查并确保 Pod.request + Quota.Allocation < Quota.Max
    - preempt: 修改并实现以下功能
        - Preemptor.Request + Quota.allocated <= Quota.min 时，表示其最小或保证资源被其他Quota使用或borrowed。 节点中的潜在受害者将从分配比其最小值更多的资源的配额中选择，即从其他配额借用资源
        - Preemptor.Request + Quota.allocated > Quota.min：表示其保证的配额没有被其他配额借用。 这样会选择属于相同配额（命名空间）且优先级低于抢占者优先级的 Pod 作为节点中的潜在受害者

- Coscheduling：https://github.com/kubernetes-sigs/scheduler-plugins/blob/master/kep/42-podgroup-coscheduling/README.md，https://github.com/kubernetes-sigs/scheduler-plugins/blob/master/kep/2-lightweight-coscheduling/README.md
  - 默认的调度器，无法保证一组Pod能够被一起调度，使用相同标签的 scheduling.x-k8s.io/pod-group 的属于同一组，计算正在运行的 pod 和正在等待的 pod（假设但未绑定）的总和，如果总和大于或等于 minMember，则将创建等待 pod。同一 PodGroup 中具有不同优先级的 Pod 可能会导致意外行为，因此需要确保同一 PodGroup 中的 Pod 具有相同的优先级
- Topology Aware：https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/kep/119-node-resource-topology-aware-scheduling， https://github.com/kubernetes-sigs/scheduler-plugins/blob/master/kep/454-numa-nodes-scoring/README.md
  - 工作节点具有不同的 NUMA 拓扑并且该拓扑中的资源量不同，Pod 可以调度在资源总量足够的节点上，但资源分配无法满足合适的 Topology 策略。和 kubelet 中的 TopologyPolicies 联合解决
- Limit Aware: https://github.com/kubernetes-sigs/scheduler-plugins/blob/master/kep/217-resource-limit-aware-scoring/README.md
  - 引入评分插件，通过跨节点“分散”或“平衡”Pod 的资源限制来缓解可突发 Pod 导致的资源过度问题
- Network-Aware: https://github.com/kubernetes-sigs/scheduler-plugins/blob/master/kep/260-network-aware-scheduling/README.md
  - 通过自定义资源 (NetworkTopology) 和 通过区域 (topology.kubernetes.io/region) 和区域 (topology.kubernetes.io/zone) 之间的权重描述底层集群的网络拓扑，通过将节点带宽容量通告为扩展资源
  - 提供 QueueSort 插件，根据 pod 的依赖关系安排在 AppGroup 中。提供网络感知的 Filter & Score 插件，根据 AppGroup 中定义的微服务依赖关系过滤和节点打分
- SySched：https://github.com/kubernetes-sigs/scheduler-plugins/blob/master/kep/399-sysched-scoring/README.md
  - 根据 pod 系统调用使用的相对风险对可行节点进行排名。 此风险计算的关键是外部系统调用 (ExS) 指标，该指标由 IBM Research 定义，用于衡量 pod 在给定节点上遭受的过多系统调用量
- Real Load Aware：https://github.com/kubernetes-sigs/scheduler-plugins/blob/master/kep/61-Trimaran-real-load-aware-scheduling/README.md
  - 通过metrics接口查询并缓存，15，10，5周期内的数据。添加过滤和打分插件
  - 过滤插件资源设置阈值，使某类资源使用率不超过 X%，通过计算公式
    - (100 - X) * (podFraction+nodeFraction) / X + X 评分
  - 打分插件会和 NodeResources 冲突，某类资源R的滑动窗口平均值M和标准差V
    - Σ((1 - min(Rm+Rv, 1.0)) * Priority) / Rlength
    - 打分插件考虑突发变化和平均值
  - 不良指标(连接失败，无响应，新添加)

        - 通过对节点评分最低来避免该节点
- Disk IO Aware: https://github.com/kubernetes-sigs/scheduler-plugins/blob/master/kep/624-disk-io-aware-scheduling/README.md
    - 磁盘的可用 IO 统计不能以与 CPU/内存统计相同的方式进行, 因此插件将使用默认值，并通过实时指标收集器获取工作负载的 IO 块大小和读/写比率
    - 对于不同厂商生产的不同类型的磁盘，磁盘 IO BW 容量与运行工作负载特性之间的数学关系有所不同，因此将为供应商提供灵活性，以 IO 驱动程序的形式插入不同的计算/标准化模型

其他调度

- 完全不同的调度器，不是 kube-scheduler 扩展，通过设置 schedulerName 使用以下可能的调度器

kube-batch

- https://github.com/kubernetes-sigs/kube-batch
- 自定义调度程序，优化批处理作业和高吞吐量计算任务的调度和执行，关键功能
    - 公平共享：确保计算资源在多个批处理作业之间公平分配
    - 基于队列的调度：将作业组织到队列中，并根据其优先级和队列要求对其进行调度
    - 作业优先级：可以为不同作业设置优先级，确保高优先级作业在低优先级作业之前调度和执行
- volcano 则基于 kube-batch 修改，具体不做多介绍
    - https://github.com/volcano-sh

Firmament

- Kubernetes 的高级调度器，引入了一种基于流网络优化的新型调度方法，旨在提高集群利用率并减少调度延迟，已经存档

openstack

- https://docs.openstack.org/nova/latest/admin/scheduling.html
- 和 kube-scheduler 不同，这只有过滤器

Yunikorn

- 提供资源调度平台，优化大规模共享集群中计算资源的分配。它旨在处理大数据工作负载、多租户环境以及其他需细粒度资源管理和调度公平性的场景。详细见
    - https://yunikorn.apache.org/docs/get_started/core_features
- 特性：分层队列、弹性资源配额和抢占等功能，调度决策基于资源请求，确保工作负载获得所需的资源
- 设计 ResourceManagerProxy，scheduler，其中 scheduler 包括多 partition，每个 partion 内包括多 queue

godel-scheduler

- https://github.com/kubewharf/godel-scheduler
- 基于apiserver开发，是一个全面的调度和资源管理平台，专为各种业务设计，以有效地运营其多样化的云原生工作负载。延续使用kube-scheduler框架，但没有使用extender
- 具有一个配额管理系统和一个管理整合资源池的调度器。在资源利用率、资源弹性和调度吞吐量方面实现更高的性能，优化集群管理的整体功能和效率。
- 在kube-scheduler基础上，加入 coscheduler ，loadAware 和 AdaptiveCpuToMemRatio 插件，详细介绍下部分插件实现
- loadAware 负载感知，针对pod和node，会使用评估器(Estimate)确定使用量和容量
  - 过滤：确保指标(profile)数据不过期，节点不超过设置的资源使用阈值
  - 评分，当前也是使用标准差方式计算，只统计 cpu 和 memory
    - (Allocat - Req) * weight * MaxScore / Allocat 其中
      - Allocat： NodeAllocate - Usage(profile)
      - Req: req * scaleFactor(由qos确定)
- AdaptiveCpuToMemRatio 是在nodeResource 加入的评分插件，和前文介绍的 LeastAllocated 同级，对节点容量引入Qos比例，并且资源使用是cpu和mem比例
  - node 比上 Pod，注意要除开边界条件，如pod请求未设置，或者节点无可分配
    - 若小于1，（node / pod +1）* 50
    - 若大于等于1，50 / (node - pod) + 1

参考

- https://www.awelm.com/posts/kube-scheduler/
  - 文中提到使用 kwok 和 scheduler-simulate 模拟测试