背景简介

什么是 Kubernetes 中的调度(Scheduling)？

Kubernetes 中的调度，是将处于待定(Pending)的 pod 绑定到某个节点的过程，由 Kubernetes 中名为 kube-scheduler 的组件执行。

调度器决定一个 pod 是否或在哪能否被调度，由调度器的可配置策略来指导，该策略包括一组规则，称为预判和优选。调度器的决定受到新 pod 第一次被调度时调度器在 Kubernetes 集群中所看到的资源情况的影响。

k8s 自身提供了很多默认调度器和一些高级的调度策略(如 nodeAffinity 等)，基本上可以满足大部分业务需求，对于特定业务，我们还可以自定义调度器（使用 python/shell/go 等实现）。

由于 Kubernetes 集群是动态的且状态会发生变化，有时候我们希望将已经运行的 pod 调度到其他节点，原因如下：

- 一些节点资源利用不足或使用过度；
- 原始调度策略不再适用，比如节点添加或删除了污点或标签，不再满足 pod/node 亲和性要求；
- 某些节点发生故障，其上 pod 被调度至其他节点；
- 有新节点被添加到群集中；
- 其它

这些问题可能引起当前调度器分配 pod 不均衡，造成节点性能瓶颈掉线，最终引发集群雪崩。
我们大致从如下可能的点进行优化：

- limit：限制每个 pod 的资源分配，设置对 cpu/mem/disk 的 requests 和 limits；
- type：为每台节点指定类型或标签，可用于作为调度算法的参数。如 CPU/内存/IO 密集或高网络带宽使用等；
- 重平衡：采取人为介入或定时任务的方式，根据多种维度去对当前的 pod 分布做重平衡；
- 节点层面的优化，以及 pod 层面的优化；

调研文档

- 调度器性能调优
  https://kubernetes.io/docs/concepts/configuration/scheduler-perf-tuning

- 优化预选阶段，丢弃不满足条件的 node
  https://zhuanlan.zhihu.com/p/33470869
  添加 alwaysCheckAllPredicates 开关，在预选阶段不符合条件的候选 Node，将不会参与后续条件的判断。

- k8s 调度详解(预选、优选过程)
  http://dockone.io/article/2885
  nodeAffinity 具备 nodeSelector 的全部功能，所以未来 Kubernetes 会将 nodeSelector 废除

- 活用 k8s 的默认调度器和一些高级调度策略，特殊业务可自定义调度器(python/shell/go)
  http://www.rhca.me/?p=44

- 使用 Kubernetes 实现高级的调度技巧
  https://thenewstack.io/implementing-advanced-scheduling-techniques-with-kubernetes

- Kubernetes 是如何转变成一个通用调度器的
  https://thenewstack.io/how-kubernetes-is-transforming-into-a-universal-scheduler

- 根据 nodeSelector 或 Affinity 为 pod 指派 node
  https://kubernetes.io/docs/concepts/configuration/assign-pod-node

- Kubernetes最佳实践:资源需求和上限
  https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-resource-requests-and-limits

- 管理容器的计算资源
  https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container

- 为容器和容器组分配内存资源
  https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource

- 为 Namespace 配置默认的 CPU/内存 需求和上限
  https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/cpu-default-namespace
  https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-default-namespace

- 为 Namespace 配置最小和最大 CPU/内存 限制
  https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/cpu-constraint-namespace

https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-constraint-namespace

- 为 Namespace 配置 内存和CPU 配额
  https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/quota-memory-cpu-namespace

- 为 Namespace 配置 pod 限额(数量)
  https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/quota-pod-namespace

应用实践

调度器在调度过程中是如何计算哪些节点是满足**可调度**条件的呢？

首先是判断节点是否有可分配的资源，如果有多台满足**可调度**条件，则调度至资源最优的节点上。

Kubernetes 每个节点上的 kubelet 会监控该节点的 **资源容量** 和 **可用资源**，我们可以通过下面的命令查看：

```
# kubectl describe node "node01" | grep -A16 "Addresses:"
Addresses:
  InternalIP:  172.xx.xxx.xx
  Hostname:    node01
Capacity:
 cpu:                4
 ephemeral-storage:  41926416Ki
 hugepages-1Gi:      0
 hugepages-2Mi:      0
 memory:             8010544Ki
 pods:               110
Allocatable:
 cpu:                3800m
 ephemeral-storage:  38639384922
 hugepages-1Gi:      0
 hugepages-2Mi:      0
 memory:             7408144Ki
 pods:               110
```

其中，`Capacity` 表示节点的资源容量，即机器的 CPU、内存、磁盘总量。本地磁盘被定义为 **短暂存储**，生命周期与 pod 一致。它与 **持久存储** 相对应。

`Allocatable` 则表示节点的可用资源，它是除开 **系统预留** 和 **k8s 预留** 以外的资源，是可以分配给集群创建 pod 使用的资源。

每个 pod 或容器在创建时都应该声明自己所需资源的 **请求值**(requests) 和 **上限值**(limits)，以便 kubelet 计算和统计整个节点资源的使用量和百分比。可以使用下面命令来查看当前节点的资源使用情况：

```
# kubectl describe node "node03" | sed '1,/Non-terminated Pods/d'
  Namespace                  Name                    CPU Requests  CPU Limits  Me
  ---------                  ----                    ------------  ----------  --
  default                    c3-registercenter-0     0 (0%)        0 (0%)      0
  kube-system                calico-node-dvmgv       150m (3%)     300m (7%)   64
  kube-system                kube-proxy-node03       150m (3%)     500m (12%)  64
  kube-system                nginx-proxy-node03      25m (0%)      300m (7%)   32
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests  CPU Limits   Memory Requests  Memory Limits
  ------------  ----------   ---------------  -------------
  325m (8%)     1100m (28%)  160M (2%)        3012M (38%)
```

这里可以看到每个 pod 有单独的资源使用统计，最后还有汇总。它是如何统计的呢？

可以发现它只是简单的相加，这些统计是针对 `limits` 和 `requests` 的，并不是 pod 运行中实际使用的资源。也就是说如果一个 pod 没有设置 limits 和 requests(参考上面的 `0 (0%)`)，那么它就可以无限使用资源，还不会被纳入统计，这就使得统计结果不准确，可能节点已经超负荷运行了，但 Sheduler 还认为它有可用资源，直至节点被拖死，集群发生雪崩。
因此，我们应该采取一些合理的手段来避免这种情况：

- 在创建 pod 时，主动为 pod 提交资源申请和限制；
- 让 sheduler 以指标采集工具采集的指标为准来计算可用资源；
- 为系统 和 k8s 预留适当的资源；

下面介绍一下资源预留和应用场景。

```
      Node Capacity
---------------------------
|     kube-reserved       |
|-------------------------|
|     system-reserved     |
|-------------------------|
|    eviction-threshold   |
|-------------------------|
|                         |
|      allocatable        |
```

```
|   (available for pods)  |
|                         |
---------------------------
```

上图表示了一个节点上资源的分配关系。

- Node Capacity
  代表整个节点的资源容量，和 `kubectl describe node node01` 看到的 Capacity 一致。

- kube-reserved
  a.) 为 k8s 后台服务预留资源，如 `kubelet`，`container runtime`，`node problem detector` 等，但并不包含那些运行在 pod 中的 k8s 服务，如 `calico-node`、`kube-apiserver`、`kube-controller-manager`、`kube-dns`、`kube-proxy`、`kube-scheduler`、`kube-scheduler`、`kubedns-autoscaler`、`kubernetes-dashboard` 等。
  b.) 在 systemd 系统上建议将的 k8s 后台服务放置在顶级 cgroup 下(设置 runtime.slice)；
  c.) 在 kubelet 服务启动参数中添加标记来设置，示例：
  `--kube-reserved=[cpu=100m][,][memory=100Mi][,][ephemeral-storage=1Gi]`

- system-reserved
  a.) 为系统后台服务预留资源，如 `sshd`、`udev` 等；
  b.) 推荐为内核预留内存资源因为目前为止它还没有被计入 k8s 的 pod 资源；
  c.) 推荐为 systemd 系统的**用户登录会话**预留资源(cgroup 设置 user.slice)；
  d.) 在 kubelet 服务启动参数中添加标记来设置，示例：
  `--system-reserved=[cpu=100m][,][memory=100Mi][,][ephemeral-storage=1Gi]`

- eviction-threshold
  a.) 为 k8s 节点设置空余资源阈值，当节点可用资源低于该阈值时，pod 将被驱逐；
  b.) 目前仅适用于 memory 和 ephemeral-storage；
  c.) 在 kubelet 服务启动参数中添加标记来设置，示例：
  `--eviction-hard=[memory.available<500Mi]`

关于资源预留，参考文档：
https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources

重平衡工具 Descheduler

本文着重介绍一下 Kubernetes 孵化器中的一个工具：Descheduler

工具简介

Descheduler 的出现就是为了解决 Kubernetes 自身调度（一次性调度）不足的问题。它以定时任务方式运行，根据已实现的策略，重新去平衡 pod 在集群中的分布。

截止目前，Descheduler 已实现的策略和计划中的功能点如下：

已实现的调度策略

- RemoveDuplicates
  移除重复 pod
- LowNodeUtilization
  节点低度使用
- RemovePodsViolatingInterPodAntiAffinity
  移除违反pod反亲和性的 pod
- RemovePodsViolatingNodeAffinity

路线图中计划实现的功能点

- Strategy to consider taints and tolerations
  考虑污点和容忍
- Consideration of pod affinity
  考虑 pod 亲和性
- Strategy to consider pod life time
  考虑 pod 生命周期
- Strategy to consider number of pending pods
  考虑待定中的 pod 数量
- Integration with cluster autoscaler
  与集群自动伸缩集成
- Integration with metrics providers for obtaining real load metrics
  与监控工具集成来获取真正的负载指标
- Consideration of Kubernetes's scheduler's predicates
  考虑 k8s 调度器的预判机制

策略介绍

- RemoveDuplicates
  此策略确保每个副本集(RS)、副本控制器(RC)、部署(Deployment)或任务(Job)只有一个 pod 被分配到同一台 node 节点上。如果有多个则会被驱逐到其它节点以便更好的在集群内分散 pod。

- LowNodeUtilization
  a. 此策略会找到未充分使用的 node 节点并在可能的情况下将那些被驱逐后希望重建的

pod 调度到该节点上。
b. 节点是否**利用不足**由一组可配置的 **阈值**(`thresholds`) 决定。这组阈值是以百分比方式指定了 CPU、内存以及 pod数量 的。只有当所有被评估资源都低于它们的阈值时，该 node 节点才会被认为处于**利用不足**状态。
c. 同时还存在一个 **目标阈值**(`targetThresholds`)，用于评估那些节点是否因为超出了阈值而应该从其上驱逐 pod。任何阈值介于 `thresholds` 和 `targetThresholds` 之间的节点都被认为资源被合理利用了，因此不会发生 pod 驱逐行为（无论是被驱逐走还是被驱逐来）。
d. 与之相关的还有另一个参数 `numberOfNodes` ，这个参数用来激活指定数量的节点是否处于资源**利用不足**状态而发生 pod 驱逐行为。

- RemovePodsViolatingInterPodAntiAffinity
  此策略会确保 node 节点上违反 pod 间亲和性的 pod 被驱逐。比如节点上有 podA 并且 podB 和 podC（也在同一节点上运行）具有禁止和 podA 在同一节点上运行的反亲和性规则，则 podA 将被从节点上驱逐，以便让 podB 和 podC 可以运行。

- RemovePodsViolatingNodeAffinity
  此策略会确保那些违反 node 亲和性的 pod 被驱逐。比如 podA 运行在 nodeA 上，后来该节点不再满足 podA 的 node 亲和性要求，如果此时存在 nodeB 满足这一要求，则 podA 会被驱逐到 nodeB 节点上。

遵循机制

当 Descheduler 调度器决定于驱逐 pod 时，它将遵循下面的机制：

- Critical pods (with annotations scheduler.alpha.kubernetes.io/critical-pod) are never evicted
  关键 pod（带注释 scheduler.alpha.kubernetes.io/critical-pod ）永远不会被驱逐。

- Pods (static or mirrored pods or stand alone pods) not part of an RC, RS, Deployment or Jobs are never evicted because these pods won't be recreated
  不属于RC，RS，部署或作业的Pod（静态或镜像pod或独立pod）永远不会被驱逐，因为这些pod不会被重新创建。

- Pods associated with DaemonSets are never evicted
  与 DaemonSets 关联的 Pod 永远不会被驱逐。

- Pods with local storage are never evicted
  具有本地存储的 Pod 永远不会被驱逐。

- BestEffort pods are evicted before Burstable and Guaranteed pods
  QoS 等级为 BestEffort 的 pod 将会在等级为 Burstable 和 Guaranteed 的 pod 之前被驱

逐。

工具使用

Descheduler 会以 Job 形式在 pod 内运行，因为 Job 具有多次运行而无需人为介入的优势。为了避免被自己驱逐 Descheduler 将会以 **关键型** pod 运行，因此它只能被创建建到 `kube-system` namespace 内。
关于 Critical pod 的介绍请参考：Guaranteed Scheduling For Critical Add-On Pods

要使用 Descheduler，我们需要编译该工具并构建 Docker 镜像，创建 ClusterRole、ServiceAccount、ClusterRoleBinding、ConfigMap 以及 Job。

由于文档中有一些小问题，手动执行这些步骤不会很顺利，我们推荐使用有人维护的现成 helm charts。
项目地址：https://github.com/komljen/helm-charts/tree/master/descheduler
使用方式：

```
helm repo add akomljen-charts \
     https://raw.githubusercontent.com/komljen/helm-charts/master/charts/

helm install \
     --name ds \
     --namespace kube-system \
     akomljen-charts/descheduler
```

该 Chart 默认设置如下：

- Descheduler 以 CronJob 方式运行，每 30 分钟执行一次，可根据实际需要进行调整；
- 在 ConfigMap 中同时内置了 4种策略，可根据实际需要禁用或调整；

值得注意的是，Descheduler 项目文档中是以 Job 方式运行，属于一次性任务。

问题处理

在手动编译构建 Descheduler 过程中，我们发现官方文档有个小问题。
比如，文档中 Job 的启动方式如下：

```
    command:
    - "/bin/sh"
    - "-ec"
    - |
   /bin/descheduler --policy-config-file /policy-dir/policy.yaml
```

即，指定使用 `sh` 来运行 `descheduler`，然而该工具的 Dockerfile 却是以白手起家（FROM scratch）方式添加 descheduler 的编译产物到容器内，新容器并不包含工具 `sh`, 这就导致 Job 在运行后因为没有 `sh` 而失败并不断拉起新的 Job，短时间内大量 Job 被创建, node 节点资源也逐渐被消耗。

关于该问题我提交了一个 isusse #133，解决办法有2种：

- 从 command 中移除 `/bin/sh -ec`
- 修改 Dockerfile 中 `From scratch` 为 `From alpine` 或 busybox 或 centos 等

然后重新构建并创建 Job。

参考文档

https://akomljen.com/meet-a-kubernetes-descheduler