42 | Kubernetes默认调度器调度策略解析

42 | Kubernetes默认调度器调度策略解析

张磊 2018-11-28

11:03

讲述：张磊 大小：5.07M

你好，我是张磊。今天我和你分享的主题是：Kubernetes 默认调度器调度策略解析。

在上一篇文章中，我主要为你讲解了 Kubernetes 默认调度器的设计原理和架构。在今天这篇文章中，我们就专注在调度过程中 Predicates 和 Priorities 这两个调度策略主要发生作用的阶段。

首先，我们一起看看 Predicates。

Predicates 在调度过程中的作用，可以理解为 Filter，即：它按照调度策略，从当前集群的所有节点中，“过滤”出一系列符合条件的节点。这些节点，都是可以运行待调度 Pod 的宿主机。

而在 Kubernetes 中，默认的调度策略有如下三种。

第一种类型，叫作 GeneralPredicates。

顾名思义，这一组过滤规则，负责的是最基础的调度策略。比如，PodFitsResources 计算的就是宿主机的 CPU 和内存资源等是否够用。

当然，我在前面已经提到过，PodFitsResources 检查的只是 Pod 的 requests 字段。需要注意的是，Kubernetes 的调度器并没有为 GPU 等硬件资源定义具体的资源类型，而是统一用一种名叫 Extended Resource 的、Key-Value 格式的扩展字段来描述的。比如下面这个例子：

```
apiVersion: v1
kind: Pod
metadata:
name: extended-resource-demo
spec:
containers:
- name: extended-resource-demo-ctr
image: nginx
resources:
requests:
alpha.kubernetes.io/nvidia-gpu: 2
limits:
alpha.kubernetes.io/nvidia-gpu: 2
```

□复制代码

可以看到，我们这个 Pod 通过alpha.kubernetes.io/nvidia-gpu=2这样的定义方式，声明使用了两个 NVIDIA 类型的 GPU。

而在 PodFitsResources 里面，调度器其实并不知道这个字段 Key 的含义是 GPU，而是直接使用后面的 Value 进行计算。当然，在 Node 的 Capacity 字段里，你也得相应地加上这台宿主机上 GPU 的总数，比如：alpha.kubernetes.io/nvidia-gpu=4。这些流程，我在后面讲解 Device Plugin 的时候会详细介绍到。

而 PodFitsHost 检查的是，宿主机的名字是否跟 Pod 的 spec.nodeName 一致。

PodFitsHostPorts 检查的是，Pod 申请的宿主机端口（spec.nodePort）是不是跟已经被使用的端口有冲突。

PodMatchNodeSelector 检查的是，Pod 的 nodeSelector 或者 nodeAffinity 指定的节点，是否与待考察节点匹配，等等。

可以看到，像上面这样一组 GeneralPredicates，正是 Kubernetes 考察一个 Pod 能不能运行在一个 Node 上最基本的过滤条件。所以，GeneralPredicates 也会被其他组件（比如 kubelet）直接调用。

我在上一篇文章中已经提到过，kubelet 在启动 Pod 前，会执行一个 Admit 操作来进行二次确认。这里二次确认的规则，就是执行一遍 GeneralPredicates。

第二种类型，是与 Volume 相关的过滤规则。

这一组过滤规则，负责的是跟容器持久化 Volume 相关的调度策略。

其中，NoDiskConflict 检查的条件，是多个 Pod 声明挂载的持久化 Volume 是否有冲突。比如，AWS EBS 类型的 Volume，是不允许被两个 Pod 同时使用的。所以，当一个名叫 A 的 EBS Volume 已经被挂载在了某个节点上时，另一个同样声明使用这个 A Volume 的 Pod，就不能被调度到这个节点上了。

而 MaxPDVolumeCountPredicate 检查的条件，则是一个节点上某种类型的持久化 Volume 是不是已经超过了一定数目，如果是的话，那么声明使用该类型持久化 Volume 的 Pod 就不能再调度到这个节点了。

而 VolumeZonePredicate，则是检查持久化 Volume 的 Zone（高可用域）标签，是否与待考察节点的 Zone 标签相匹配。

此外，这里还有一个叫作 VolumeBindingPredicate 的规则。它负责检查的，是该 Pod 对应的 PV 的 nodeAffinity 字段，是否跟某个节点的标签相匹配。

在前面的第 29 篇文章《PV、PVC 体系是不是多此一举？从本地持久化卷谈起》中，我曾经为你讲解过，Local Persistent Volume（本地持久化卷），必须使用 nodeAffinity 来跟某个具体的节点绑定。这其实也就意味着，在 Predicates 阶段，Kubernetes 就必须能够根据 Pod 的 Volume 属性来进行调度。

此外，如果该 Pod 的 PVC 还没有跟具体的 PV 绑定的话，调度器还要负责检查所有待绑定 PV，当有可用的 PV 存在并且该 PV 的 nodeAffinity 与待考察节点一致时，这条规则才会返回“成功”。比如下面这个例子：

```
apiVersion: v1
kind: PersistentVolume
```

```
metadata:
name: example-local-pv
spec:
capacity:
storage: 500Gi
accessModes:
- ReadWriteOnce
persistentVolumeReclaimPolicy: Retain
storageClassName: local-storage
local:
path: /mnt/disks/vol1
nodeAffinity:
required:
nodeSelectorTerms:
- matchExpressions:
- key: kubernetes.io/hostname
operator: In
values:
- my-node
```

□复制代码

可以看到，这个 PV 对应的持久化目录，只会出现在名叫 my-node 的宿主机上。所以，任何一个通过 PVC 使用这个 PV 的 Pod，都必须被调度到 my-node 上才可以正常工作。VolumeBindingPredicate，正是调度器里完成这个决策的位置。

第三种类型，是宿主机相关的过滤规则。

这一组规则，主要考察待调度 Pod 是否满足 Node 本身的某些条件。

比如，PodToleratesNodeTaints，负责检查的就是我们前面经常用到的 Node 的“污点”机制。只有当 Pod 的 Toleration 字段与 Node 的 Taint 字段能够匹配的时候，这个 Pod 才能被调度到该节点上。

> *备注：这里，你也可以再回顾下第 21 篇文章《容器化守护进程的意义：DaemonSet》中的相关内容。*

而 NodeMemoryPressurePredicate，检查的是当前节点的内存是不是已经不够充足，如果是的话，那么待调度 Pod 就不能被调度到该节点上。

第四种类型，是 Pod 相关的过滤规则。

这一组规则，跟 GeneralPredicates 大多数是重合的。而比较特殊的，是 PodAffinityPredicate。这个规则的作用，是检查待调度 Pod 与 Node 上的已有 Pod 之间的亲密（affinity）和反亲密（anti-affinity）关系。比如下面这个例子：

```
apiVersion: v1
kind: Pod
metadata:
name: with-pod-antiaffinity
spec:
affinity:
podAntiAffinity:
requiredDuringSchedulingIgnoredDuringExecution:
- weight: 100
podAffinityTerm:
labelSelector:
matchExpressions:
- key: security
operator: In
values:
- S2
topologyKey: kubernetes.io/hostname
containers:
- name: with-pod-affinity
image: docker.io/ocpqe/hello-pod
```

复制代码

这个例子里的 podAntiAffinity 规则，就指定了这个 Pod 不希望跟任何携带了 security=S2 标签的 Pod 存在于同一个 Node 上。需要注意的是，PodAffinityPredicate 是有作用域的，比如上面这条规则，就仅对携带了 Key 是 kubernetes.io/hostname标签的 Node 有效。这正是 topologyKey 这个关键词的作用。

而与 podAntiAffinity 相反的，就是 podAffinity，比如下面这个例子：

```
apiVersion: v1
kind: Pod
metadata:
name: with-pod-affinity
spec:
affinity:
podAffinity:
requiredDuringSchedulingIgnoredDuringExecution:
- labelSelector:
matchExpressions:
- key: security
operator: In
values:
- S1
topologyKey: failure-domain.beta.kubernetes.io/zone
containers:
- name: with-pod-affinity
image: docker.io/ocpqe/hello-pod
```

□复制代码

这个例子里的 Pod，就只会被调度到已经有携带了 security=S1 标签的 Pod 运行的 Node 上。而这条规则的作用域，则是所有携带 Key 是failure-domain.beta.kubernetes.io/zone标签的 Node。

此外，上面这两个例子里的 requiredDuringSchedulingIgnoredDuringExecution 字段的含义是：这条规则必须在 Pod 调度时进行检查（requiredDuringScheduling）；但是如果是已经在运行的 Pod 发生变化，比如 Label 被修改，造成了该 Pod 不再适合运行在这个 Node 上的时候，Kubernetes 不会进行主动修正（IgnoredDuringExecution）。

上面这四种类型的 Predicates，就构成了调度器确定一个 Node 可以运行待调度 Pod 的基本策略。

在具体执行的时候， 当开始调度一个 Pod 时，Kubernetes 调度器会同时启动 16 个 Goroutine，来并发地为集群里的所有 Node 计算 Predicates，最后返回可以运行这个 Pod 的宿主机列表。

需要注意的是，在为每个 Node 执行 Predicates 时，调度器会按照固定的顺序来进行检查。这个顺序，是按照 Predicates 本身的含义来确定的。比如，宿主机相关的 Predicates 会被放在相对靠前的位置进行检查。要不然的话，在一台资源已经严重不足的宿主机上，上来就开始计算 PodAffinityPredicate，是没有实际意义的。

接下来，我们再来看一下 Priorities。

在 Predicates 阶段完成了节点的“过滤”之后，Priorities 阶段的工作就是为这些节点打分。这里打分的范围是 0-10 分，得分最高的节点就是最后被 Pod 绑定的最佳节点。

Priorities 里最常用到的一个打分规则，是 LeastRequestedPriority。它的计算方法，可以简单地总结为如下所示的公式：

```
score = (cpu((capacity-sum(requested))10/capacity) +
memory((capacity-sum(requested))10/capacity))/2
```

复制代码

可以看到，这个算法实际上就是在选择空闲资源（CPU 和 Memory）最多的宿主机。

而与 LeastRequestedPriority 一起发挥作用的，还有 BalancedResourceAllocation。它的计算公式如下所示：

```
score = 10 - variance(cpuFraction,memoryFraction,volumeFraction)*10
```

复制代码

其中，每种资源的 Fraction 的定义是 ： Pod 请求的资源 / 节点上的可用资源。而 variance 算法的作用，则是计算每两种资源 Fraction 之间的“距离”。而最后选择的，则是资源 Fraction 差距最小的节点。

所以说，BalancedResourceAllocation 选择的，其实是调度完成后，所有节点里各种资源分配最均衡的那个节点，从而避免一个节点上 CPU 被大量分配、而 Memory 大量剩余的情况。

此外，还有 NodeAffinityPriority、TaintTolerationPriority 和 InterPodAffinityPriority 这三种 Priority。顾名思义，它们与前面的 PodMatchNodeSelector、PodToleratesNodeTaints 和 PodAffinityPredicate 这三个 Predicate 的含义和计算方法是类似的。但是作为 Priority，一个 Node 满足上述规则的字段数目越多，它的得分就会越高。

在默认 Priorities 里，还有一个叫作 ImageLocalityPriority 的策略。它是在 Kubernetes v1.12 里新开启的调度规则，即：如果待调度 Pod 需要使用的镜像

很大，并且已经存在于某些 Node 上，那么这些 Node 的得分就会比较高。

当然，为了避免这个算法引发调度堆叠，调度器在计算得分的时候还会根据镜像的分布进行优化，即：如果大镜像分布的节点数目很少，那么这些节点的权重就会被调低，从而“对冲”掉引起调度堆叠的风险。

以上，就是 Kubernetes 调度器的 Predicates 和 Priorities 里默认调度规则的主要工作原理了。

在实际的执行过程中，调度器里关于集群和 Pod 的信息都已经缓存化，所以这些算法的执行过程还是比较快的。

此外，对于比较复杂的调度算法来说，比如 PodAffinityPredicate，它们在计算的时候不只关注待调度 Pod 和待考察 Node，还需要关注整个集群的信息，比如，遍历所有节点，读取它们的 Labels。这时候，Kubernetes 调度器会在为每个待调度 Pod 执行该调度算法之前，先将算法需要的集群信息初步计算一遍，然后缓存起来。这样，在真正执行该算法的时候，调度器只需要读取缓存信息进行计算即可，从而避免了为每个 Node 计算 Predicates 的时候反复获取和计算整个集群的信息。

总结

在本篇文章中，我为你讲述了 Kubernetes 默认调度器里的主要调度算法。

需要注意的是，除了本篇讲述的这些规则，Kubernetes 调度器里其实还有一些默认不会开启的策略。你可以通过为 kube-scheduler 指定一个配置文件或者创建一个 ConfigMap ，来配置哪些规则需要开启、哪些规则需要关闭。并且，你可以通过为 Priorities 设置权重，来控制调度器的调度行为。

思考题

请问，如何能够让 Kubernetes 的调度器尽可能地将 Pod 分布在不同机器上，避免“堆叠”呢？请简单描述下你的算法。