**加我微信可进群学习交流：**

**微信号：**

luckylucky421302

也可通过扫描下面二维码添加



**课程更新的知识点会通过微信公众号免费分享给大家，可以关注我的公众号**

****

**官方解释：**

节点亲和性，是*pod*的一种属性（偏好或硬性要求），它使*pod*被吸引到一类特定的节点。Taint则相反，它使*节点*能够*排斥*一类特定的 pod。Taint 和 toleration 相互配合，可以用来避免pod 被分配到不合适的节点上。每个节点上都可以应用一个或多个 taint ，这表示对于那些不能容忍这些 taint 的 pod，是不会被该节点接受的。如果将 toleration 应用于 pod 上，则表示这些 pod 可以（但不要求）被调度到具有匹配 taint 的节点上。

**自我理解：**

污点和容忍度是相互匹配的关系，我们在node上定义一个污点，pod上定义容忍度，如果pod能容忍这个污点，就会被调度到拥有这个污点的节点上，不能容忍这个污点就不会调度到拥有这个污点的节点上，如果node节点上没有定义污点，那么任何pod都会调度到这个节点上

我们可以给node节点打一个污点，pod如果不能容忍这个节点上定义的污点，那就调度不到这个节点上，污点是定义在节点上的key value这种键值数据

taints：污点，定义在节点上，是键值数据

tolerations：容忍度，定义在pod上，可以定义能容忍哪些污点

**概念：**

你可以使用命令 [kubectl taint](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands" \l "taint) 给节点增加一个 taint。比如，

kubectl taint nodes node1 key=value:NoSchedule

给节点 node1 增加一个 taint，它的 key 是 key，value 是 value，effect 是 NoSchedule。这表示只有拥有和这个 taint 相匹配的 toleration 的 pod 才能够被分配到 node1 这个节点。你可以在 PodSpec 中定义 pod 的 toleration。下面两个 toleration 均与上面例子中使用 kubectl taint 命令创建的 taint 相匹配，因此如果一个 pod 拥有其中的任何一个 toleration 都能够被分配到 node1 ：

想删除上述命令添加的 taint ，你可以运行：

kubectl taint nodes node1 key:NoSchedule-

你可以在 PodSpec 中为容器设定容忍标签。以下两个容忍标签都与上面的 kubectl taint 创建的污点“匹配”， 因此具有任一容忍标签的Pod都可以将其调度到 node1 上：

**tolerations**:

- **key**: "key"

**operator**: "Equal"

**value**: "value"

**effect**: "NoSchedule"

**tolerations**:

- **key**: "key"

**operator**: "Exists"

**effect**: "NoSchedule"

一个 toleration和一个 taint 相“匹配”是指它们有一样的key和effect ，并且：

* 如果 operator 是 Exists （此时 toleration 不能指定 value），或者
* 如果 operator 是 Equal ，则它们的 value 应该相等

**注意：**

* 如果一个 toleration 的 effect 为空，则 key 值与之相同的相匹配 taint 的 effect 可以是任意值。

**tolerations**:

- **key**: "key"

**operator**: "Exists"

上述例子使用到的 effect 的一个值 NoSchedule，你也可以使用另外一个值 PreferNoSchedule。这是“优化”或“软”版本的 NoSchedule ——系统会 *尽量* 避免将 pod 调度到存在其不能容忍 taint 的节点上，但这不是强制的。effect 的值还可以设置为 NoExecute，下文会详细描述这个值。

你可以给一个节点添加多个 taint ，也可以给一个 pod 添加多个 toleration。Kubernetes 处理多个 taint 和 toleration 的过程就像一个过滤器：从一个节点的所有 taint 开始遍历，过滤掉那些 pod 中存在与之相匹配的 toleration 的 taint。余下未被过滤的 taint 的 effect 值决定了 pod 是否会被分配到该节点，特别是以下情况：

* 如果未被过滤的 taint 中存在一个以上 effect 值为 NoSchedule 的 taint，则 Kubernetes 不会将 pod 分配到该节点。
* 如果未被过滤的 taint 中不存在 effect 值为 NoSchedule 的 taint，但是存在 effect 值为 PreferNoSchedule 的 taint，则 Kubernetes 会 *尝试* 将 pod 分配到该节点。
* 如果未被过滤的 taint 中存在一个以上 effect 值为 NoExecute 的 taint，则 Kubernetes 不会将 pod 分配到该节点（如果 pod 还未在节点上运行），或者将 pod 从该节点驱逐（如果 pod 已经在节点上运行）。

例如，假设您给一个节点添加了如下的 taint

kubectl taint nodes node1 key1=value1:NoSchedule

kubectl taint nodes node1 key1=value1:NoExecute

kubectl taint nodes node1 key2=value2:NoSchedule

然后存在一个 pod，它有两个 toleration：

**tolerations**:

- **key**: "key1"

**operator**: "Equal"

**value**: "value1"

**effect**: "NoSchedule"

- **key**: "key1"

**operator**: "Equal"

**value**: "value1"

**effect**: "NoExecute"

在这个例子中，上述 pod 不会被分配到上述节点，因为其没有 toleration 和第三个 taint 相匹配。但是如果在给节点添加上述 taint 之前，该 pod 已经在上述节点运行，那么它还可以继续运行在该节点上，因为第三个 taint 是三个 taint 中唯一不能被这个 pod 容忍的。通常情况下，如果给一个节点添加了一个 effect 值为 NoExecute 的 taint，则任何不能忍受这个 taint 的 pod 都会马上被驱逐，任何可以忍受这个 taint 的 pod 都不会被驱逐。但是，如果 pod 存在一个 effect 值为 NoExecute 的 toleration 指定了可选属性 tolerationSeconds 的值，则表示在给节点添加了上述 taint 之后，pod 还能继续在节点上运行的时间。例如，

**tolerations**:

- **key**: "key1"

**operator**: "Equal"

**value**: "value1"

**effect**: "NoExecute"

**tolerationSeconds**: 3600

这表示如果这个 pod 正在运行，然后一个匹配的 taint 被添加到其所在的节点，那么 pod 还将继续在节点上运行 3600 秒，然后被驱逐。如果在此之前上述 taint 被删除了，则 pod 不会被驱逐。

使用例子

通过 taint 和 toleration，可以灵活地让 pod *避开* 某些节点或者将 pod 从某些节点驱逐。下面是几个使用例子：

* **专用节点**：如果你想将某些节点专门分配给特定的一组用户使用，你可以给这些节点添加一个 taint（即， kubectl taint nodes nodename dedicated=groupName:NoSchedule），然后给这组用户的 pod 添加一个相对应的 toleration（通过编写一个自定义的 [admission controller](https://kubernetes.io/docs/admin/admission-controllers/)，很容易就能做到）。拥有上述 toleration 的 pod 就能够被分配到上述专用节点，同时也能够被分配到集群中的其它节点。如果你希望这些 pod 只能被分配到上述专用节点，那么你还需要给这些专用节点另外添加一个和上述 taint 类似的 label （例如：dedicated=groupName），同时 还要在上述 admission controller 中给 pod 增加节点亲和性要求上述 pod 只能被分配到添加了 dedicated=groupName 标签的节点上。
* **配备了特殊硬件的节点**：在部分节点配备了特殊硬件（比如 GPU）的集群中，我们希望不需要这类硬件的 pod 不要被分配到这些特殊节点，以便为后继需要这类硬件的 pod 保留资源。要达到这个目的，可以先给配备了特殊硬件的节点添加 taint（例如 kubectl taint nodes nodename special=true:NoSchedule or kubectl taint nodes nodename special=true:PreferNoSchedule)，然后给使用了这类特殊硬件的 pod 添加一个相匹配的 toleration。和专用节点的例子类似，添加这个 toleration 的最简单的方法是使用自定义 [admission controller](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/)。比如，我们推荐使用 [Extended Resources](https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/#extended-resources) 来表示特殊硬件，给配置了特殊硬件的节点添加 taint 时包含 extended resource 名称，然后运行一个 [ExtendedResourceToleration](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/" \l "extendedresourcetoleration) admission controller。此时，因为节点已经被 打上taint 了，没有对应 toleration 的 Pod 会被调度到这些节点。但当你创建一个使用了 extended resource 的 Pod 时，ExtendedResourceToleration admission controller 会自动给 Pod 加上正确的 toleration ，这样 Pod 就会被自动调度到这些配置了特殊硬件件的节点上。这样就能够确保这些配置了特殊硬件的节点专门用于运行 需要使用这些硬件的 Pod，并且你无需手动给这些 Pod 添加 toleration。
* **基于 taint 的驱逐**: 这是在每个 pod 中配置的在节点出现问题时的驱逐行为，接下来的章节会描述这个特性

基于 taint 的驱逐

**FEATURE STATE:** Kubernetes v1.18 [stable]

前文我们提到过 taint 的 effect 值 NoExecute ，它会影响已经在节点上运行的 pod

* 如果 pod 不能忍受 effect 值为 NoExecute 的 taint，那么 pod 将马上被驱逐
* 如果 pod 能够忍受 effect 值为 NoExecute 的 taint，但是在 toleration 定义中没有指定 tolerationSeconds，则 pod 还会一直在这个节点上运行。
* 如果 pod 不能够忍受 effect 值为 NoExecute 的 taint，而且指定了 tolerationSeconds，则 pod 还能在这个节点上继续运行这个指定的时间长度。

此外，Kubernetes 1.6+ 已经支持（alpha阶段）节点问题的表示。换句话说，当某种条件为真时，node controller会自动给节点添加一个 taint。当前内置的 taint 包括：

* node.kubernetes.io/not-ready：节点未准备好。这相当于节点状态 Ready 的值为 “False”。
* node.kubernetes.io/unreachable：node controller 访问不到节点. 这相当于节点状态 Ready 的值为 “Unknown”。
* node.kubernetes.io/out-of-disk：节点磁盘耗尽。
* node.kubernetes.io/memory-pressure：节点存在内存压力。
* node.kubernetes.io/disk-pressure：节点存在磁盘压力。
* node.kubernetes.io/network-unavailable：节点网络不可用。
* node.kubernetes.io/unschedulable: 节点不可调度。
* node.cloudprovider.kubernetes.io/uninitialized：如果 kubelet 启动时指定了一个 “外部” cloud provider，它将给当前节点添加一个 taint 将其标志为不可用。在 cloud-controller-manager 的一个 controller 初始化这个节点后，kubelet 将删除这个 taint。

在节点被驱逐时，节点控制器或者 kubelet 会添加带有 NoExecute 效应的相关污点。如果异常状态恢复正常，kubelet 或节点控制器能够移除相关的污点。

**注意：**

为了保证由于节点问题引起的 pod 驱逐[rate limiting](https://kubernetes.io/docs/concepts/architecture/nodes/)行为正常，系统实际上会以 rate-limited 的方式添加 taint。在像 master 和 node 通讯中断等场景下，这避免了 pod 被大量驱逐。

使用这个功能特性，结合 tolerationSeconds，pod 就可以指定当节点出现一个或全部上述问题时还将在这个节点上运行多长的时间。

比如，一个使用了很多本地状态的应用程序在网络断开时，仍然希望停留在当前节点上运行一段较长的时间，愿意等待网络恢复以避免被驱逐。在这种情况下，pod 的 toleration 可能是下面这样的：

**tolerations**:

- **key**: "node.kubernetes.io/unreachable"

**operator**: "Exists"

**effect**: "NoExecute"

**tolerationSeconds**: 6000

注意，Kubernetes 会自动给 pod 添加一个 key 为 node.kubernetes.io/not-ready 的 toleration 并配置 tolerationSeconds=300，除非用户提供的 pod 配置中已经已存在了 key 为 node.kubernetes.io/not-ready 的 toleration。同样，Kubernetes 会给 pod 添加一个 key 为 node.kubernetes.io/unreachable 的 toleration 并配置 tolerationSeconds=300，除非用户提供的 pod 配置中已经已存在了 key 为 node.kubernetes.io/unreachable 的 toleration。

这种自动添加 toleration 机制保证了在其中一种问题被检测到时 pod 默认能够继续停留在当前节点运行 5 分钟。这两个默认 toleration 是由 [DefaultTolerationSeconds admission controller](https://git.k8s.io/kubernetes/plugin/pkg/admission/defaulttolerationseconds)添加的。

[DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/) 中的 pod 被创建时，针对以下 taint 自动添加的 NoExecute 的 toleration 将不会指定 tolerationSeconds：

* node.kubernetes.io/unreachable
* node.kubernetes.io/not-ready

这保证了出现上述问题时 DaemonSet 中的 pod 永远不会被驱逐。

基于节点状态添加 taint

Node 生命周期控制器会自动创建与 Node 条件相对应的带有 NoSchedule 效应的污点。 同样，调度器不检查节点条件，而是检查节点污点。这确保了节点条件不会影响调度到节点上的内容。用户可以通过添加适当的 Pod 容忍度来选择忽略某些 Node 的问题(表示为 Node 的调度条件)。

自 Kubernetes 1.8 起， DaemonSet 控制器自动为所有守护进程添加如下 NoSchedule toleration 以防 DaemonSet 崩溃：

* node.kubernetes.io/memory-pressure
* node.kubernetes.io/disk-pressure
* node.kubernetes.io/out-of-disk (*只适合 critical pod*)
* node.kubernetes.io/unschedulable (1.10 或更高版本)
* node.kubernetes.io/network-unavailable (*只适合 host network*)

添加上述 toleration 确保了向后兼容，你也可以选择自由的向 DaemonSet 添加 toleration。

**演示说明**

kubectl describe nodes master1 查看master1信息，显示如下：

Name: master1

Roles: master

Labels: beta.kubernetes.io/arch=amd64

beta.kubernetes.io/os=linux

kubernetes.io/arch=amd64

kubernetes.io/hostname=master1

kubernetes.io/os=linux

master1=ha

node-role.kubernetes.io/master=

Annotations: kubeadm.alpha.kubernetes.io/cri-socket: /var/run/dockershim.sock

node.alpha.kubernetes.io/ttl: 0

projectcalico.org/IPv4Address: 192.168.80.170/24

volumes.kubernetes.io/controller-managed-attach-detach: true

CreationTimestamp: Sun, 07 Jul 2019 20:41:42 +0800

**Taints: node-role.kubernetes.io/master:NoSchedule**

Unschedulable: false

Conditions:

Type Status LastHeartbeatTime LastTransitionTime Reason Message

---- ------ ----------------- ------------------ ------ -------

NetworkUnavailable False Mon, 15 Jul 2019 03:19:37 +0800 Mon, 15 Jul 2019 03:19:37 +0800 CalicoIsUp Calico is running on this node

MemoryPressure False Tue, 23 Jul 2019 03:16:43 +0800 Mon, 22 Jul 2019 23:55:30 +0800 KubeletHasSufficientMemory kubelet has sufficient memory available

DiskPressure False Tue, 23 Jul 2019 03:16:43 +0800 Mon, 22 Jul 2019 23:55:30 +0800 KubeletHasNoDiskPressure kubelet has no disk pressure

PIDPressure False Tue, 23 Jul 2019 03:16:43 +0800 Mon, 22 Jul 2019 23:55:30 +0800 KubeletHasSufficientPID kubelet has sufficient PID available

Ready True Tue, 23 Jul 2019 03:16:43 +0800 Mon, 22 Jul 2019 23:55:30 +0800 KubeletReady kubelet is posting ready status

Addresses:

InternalIP: 192.168.80.180

Hostname: master1

Capacity:

cpu: 4

ephemeral-storage: 40137576Ki

hugepages-1Gi: 0

hugepages-2Mi: 0

memory: 3865308Ki

pods: 110

Allocatable:

cpu: 4

ephemeral-storage: 36990789981

hugepages-1Gi: 0

hugepages-2Mi: 0

memory: 3762908Ki

pods: 110

System Info:

Machine ID: b6c0453c62af4d9fb8494e86a543681a

System UUID: 0BF74D56-6BA1-72F5-1895-795A759D7B09

Boot ID: 376fee78-1f9b-4077-ac20-1271f2e79af3

Kernel Version: 3.10.0-693.el7.x86\_64

OS Image: CentOS Linux 7 (Core)

Operating System: linux

Architecture: amd64

Container Runtime Version: docker://18.9.3

Kubelet Version: v1.14.0

Kube-Proxy Version: v1.14.0

PodCIDR: 10.244.0.0/24

Non-terminated Pods: (10 in total)

Namespace Name CPU Requests CPU Limits Memory Requests Memory Limits AGE

--------- ---- ------------ ---------- --------------- ------------- ---

kube-system calico-node-gm28f 250m (6%) 0 (0%) 0 (0%) 0 (0%) 14h

kube-system coredns-fb8b8dccf-s6k88 100m (2%) 0 (0%) 70Mi (1%) 170Mi (4%) 7d21h

kube-system coredns-fb8b8dccf-znbpc 100m (2%) 0 (0%) 70Mi (1%) 170Mi (4%) 7d21h

kube-system etcd-master1 0 (0%) 0 (0%) 0 (0%) 0 (0%) 7d21h

kube-system kube-apiserver-master1 250m (6%) 0 (0%) 0 (0%) 0 (0%) 5h43m

kube-system kube-controller-manager-master1 200m (5%) 0 (0%) 0 (0%) 0 (0%) 7d21h

kube-system kube-proxy-gw82k 0 (0%) 0 (0%) 0 (0%) 0 (0%) 7d21h

kube-system kube-scheduler-master1 100m (2%) 0 (0%) 0 (0%) 0 (0%) 7d21h

kube-system metrics-server-6d9557dc9d-rtwp5 385m (9%) 480m (12%) 290Mi (7%) 540Mi (14%) 7d21h

kube-system traefik-ingress-controller-6827h 0 (0%) 0 (0%) 0 (0%) 0 (0%) 7d21h

Allocated resources:

(Total limits may be over 100 percent, i.e., overcommitted.)

Resource Requests Limits

-------- -------- ------

cpu 1385m (34%) 480m (12%)

memory 430Mi (11%) 880Mi (23%)

ephemeral-storage 0 (0%) 0 (0%)

Events: <none>

通过上面可以看到在master1上定义了一个污点**Taints**: node-role.kubernetes.io/master:NoSchedule

所以我们在创建pod时如果没有定义容忍度，那么就不会调度到master节点上

kubectl explain node.spec.taints

KIND: Node

VERSION: v1

RESOURCE: taints <[]Object>

DESCRIPTION:

If specified, the node's taints.

The node this Taint is attached to has the "effect" on any pod that does

not tolerate the Taint.

FIELDS:

effect <string> -required-

Required. The effect of the taint on pods that do not tolerate the taint.

Valid effects are NoSchedule, PreferNoSchedule and NoExecute.

key <string> -required-

Required. The taint key to be applied to a node.

timeAdded <string>

TimeAdded represents the time at which the taint was added. It is only

written for NoExecute taints.

value <string>

Required. The taint value corresponding to the taint key.

effect：用来定义节点污点对pod对象的排斥效果：

NoSchedule：

不允许调度，已经调度的不受影响

NoExecute：

当pod能够容忍这个节点污点，会被调度到节点，当不能容忍节点污点，将会被驱逐

PreferNoSchedule：

表示k8s将尽量避免将pod调度到这个污点的节点上，但是如果没有节点可以被调度，也是可以调度到拥有这个污点的node节点上的

kubectl describe pods calico-node-5kcv9 -n kube-system 显示如下：

Name: calico-node-5kcv9

Namespace: kube-system

Priority: 0

PriorityClassName: <none>

Node: master3/192.168.80.172

Start Time: Mon, 15 Jul 2019 11:17:43 +0800

Labels: controller-revision-hash=56b85bcd88

k8s-app=calico-node

pod-template-generation=2

Annotations: scheduler.alpha.kubernetes.io/critical-pod:

Status: Running

IP: 192.168.80.172

Controlled By: DaemonSet/calico-node

Init Containers:

install-cni:

Container ID: docker://16f8c9edfb05000ce5521945920700d114377e0901a0f74f56dcc2d79d29eaa8

Image: quay.io/calico/cni:v3.5.3

Image ID: docker://sha256:0e3609429486292a44aa8401c79e559ca55dcfa10b4d5a1cff7c81440f9be5fb

Port: <none>

Host Port: <none>

Command:

/install-cni.sh

State: Terminated

Reason: Completed

Exit Code: 0

Started: Mon, 15 Jul 2019 11:17:44 +0800

Finished: Mon, 15 Jul 2019 11:17:49 +0800

Ready: True

Restart Count: 0

Environment:

CNI\_CONF\_NAME: 10-calico.conflist

CNI\_NETWORK\_CONFIG: <set to the key 'cni\_network\_config' of config map 'calico-config'> Optional: false

KUBERNETES\_NODE\_NAME: (v1:spec.nodeName)

CNI\_MTU: <set to the key 'veth\_mtu' of config map 'calico-config'> Optional: false

SLEEP: false

Mounts:

/host/etc/cni/net.d from cni-net-dir (rw)

/host/opt/cni/bin from cni-bin-dir (rw)

/var/run/secrets/kubernetes.io/serviceaccount from calico-node-token-b9kml (ro)

Containers:

calico-node:

Container ID: docker://e079e7371dd67d11c0acc835cd07bf77358af334a17892ee7cf2b4d78e132cf4

Image: quay.io/calico/node:v3.5.3

Image ID: docker://sha256:c56b9408bf9e8e75b9cab25eeede8ebba1d491030e6123331bac70cf552f9a1d

Port: <none>

Host Port: <none>

State: Running

Started: Mon, 15 Jul 2019 11:17:51 +0800

Ready: True

Restart Count: 0

Requests:

cpu: 250m

Liveness: http-get http://localhost:9099/liveness delay=10s timeout=1s period=10s #success=1 #failure=6

Readiness: exec [/bin/calico-node -bird-ready -felix-ready] delay=0s timeout=1s period=10s #success=1 #failure=3

Environment:

DATASTORE\_TYPE: kubernetes

WAIT\_FOR\_DATASTORE: true

NODENAME: (v1:spec.nodeName)

CALICO\_NETWORKING\_BACKEND: <set to the key 'calico\_backend' of config map 'calico-config'> Optional: false

CLUSTER\_TYPE: k8s,bgp

IP: autodetect

IP\_AUTODETECTION\_METHOD: can-reach=8.8.8.8

CALICO\_IPV4POOL\_IPIP: Always

FELIX\_IPINIPMTU: <set to the key 'veth\_mtu' of config map 'calico-config'> Optional: false

CALICO\_IPV4POOL\_CIDR: 192.168.0.0/16

CALICO\_DISABLE\_FILE\_LOGGING: true

FELIX\_DEFAULTENDPOINTTOHOSTACTION: ACCEPT

FELIX\_IPV6SUPPORT: false

FELIX\_LOGSEVERITYSCREEN: info

FELIX\_HEALTHENABLED: true

Mounts:

/lib/modules from lib-modules (ro)

/run/xtables.lock from xtables-lock (rw)

/var/lib/calico from var-lib-calico (rw)

/var/run/calico from var-run-calico (rw)

/var/run/secrets/kubernetes.io/serviceaccount from calico-node-token-b9kml (ro)

Conditions:

Type Status

Initialized True

Ready True

ContainersReady True

PodScheduled True

Volumes:

lib-modules:

Type: HostPath (bare host directory volume)

Path: /lib/modules

HostPathType:

var-run-calico:

Type: HostPath (bare host directory volume)

Path: /var/run/calico

HostPathType:

var-lib-calico:

Type: HostPath (bare host directory volume)

Path: /var/lib/calico

HostPathType:

xtables-lock:

Type: HostPath (bare host directory volume)

Path: /run/xtables.lock

HostPathType: FileOrCreate

cni-bin-dir:

Type: HostPath (bare host directory volume)

Path: /opt/cni/bin

HostPathType:

cni-net-dir:

Type: HostPath (bare host directory volume)

Path: /etc/cni/net.d

HostPathType:

calico-node-token-b9kml:

Type: Secret (a volume populated by a Secret)

SecretName: calico-node-token-b9kml

Optional: false

QoS Class: Burstable

Node-Selectors: beta.kubernetes.io/os=linux

Tolerations: :NoSchedule

:NoExecute

CriticalAddonsOnly

node.kubernetes.io/disk-pressure:NoSchedule

node.kubernetes.io/memory-pressure:NoSchedule

node.kubernetes.io/network-unavailable:NoSchedule

node.kubernetes.io/not-ready:NoExecute

node.kubernetes.io/pid-pressure:NoSchedule

node.kubernetes.io/unreachable:NoExecute

node.kubernetes.io/unschedulable:NoSchedule

Events: <none>

上面的pod定义了容忍度，表示可以容忍master上定义的污点，所以会被调度到master上

**pod容忍度，node污点详细说明**

（1）可以使用kubectl taint 给节点添加污点

kubectl taint nodes node1 key=value:NoSchedule

在node1上添加了污点. 这个污点的key是 key, value是 value, 污点的effect是NoSchedule. 如果pod没有定义容忍度的话就不会调度到拥有这个污点的节点上

（2）使用下面的命令移除污点

kubectl taint nodes node1 key:NoSchedule-

你可以在pod的pod.spec字段指定容忍度. 这个容忍度必须能够容忍上面通过kubectl taint打的污点，这样pod才会调度到拥有这个污点的node上

（3）给Pod设置容忍

spec:

  tolerations: #设置容忍性

  - key: "test"

   operator: "Equal"  #如果操作符为Exists，那么value属性可省略,如果不指定operator，则默认为Equal

   value: "16"

   effect: "NoSchedule"

  #意思是这个Pod要容忍的有污点的Node的key是test，value是16,效果是NoSchedule，

  #tolerations属性下各值必须使用引号，容忍的值都是设置Node的taints时给的值。

kubectl taint nodes node1 test=16: NoSchedule

对于tolerations属性的写法：

**其中的key、value、effect 与Node的Taint设置需保持一致**， 还有以下几点说明：

         1、如果operator的值是Exists，则value属性可省略。

         2、如果operator的值是Equal，则表示其key与value之间的关系是equal(等于)。

         3、如果不指定operator属性，则默认值为Equal。

    另外，还有两个特殊值：

         1、空的key 如果再配合Exists ,就能匹配所有的key与value ，也就是能容忍所有node上的所有Taints。

         2、空的effect，匹配所有的effect

**例子，下面操作是在k8s的master节点操作**

测试effect是NoSchedule

kubectl taint nodes node1 name=lucky:NoSchedule

kubectl get pods -o wide 可以看到node1上的pod还是存在的，并且没有被驱逐

（1）定义一个没有容忍度的pod

cat pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: pod-taint

spec:

containers:

- name: nginx

image: nginx

kubectl apply -f pod.yaml

kubectl get pods 显示pod-taint处于pending状态

kubectl describe pods pod-taint 查看pod的详细信息，可看到如下提示，显示pod不能容忍节点污点，不能完成调度

0/4 nodes are available: 4 node(s) had taints that the pod didn't tolerate

（2）定义一个有容忍度的pod

cat pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: pod-taint

spec:

containers:

- name: nginx

image: nginx

tolerations:

- key: key

value: value

effect: NoSchedule

kubectl apply -f pod.yaml

kubectl get pods -o wide 可以看到pod-taint已经完成调度了，可以调度到node1节点上

删除污点：

kubectl taint nodes node1 name:NoSchedule-

修改effect为NoExecute，这步大家不要执行，知道即可：

kubectl taint nodes node1 key=value:NoExecute

kubectl get pods -o wide

发现原来已经调度到node1上的pod都被驱逐了

删除污点：

kubectl taint nodes node1 key:NoExecute-