Kubernetes应用部署策略实践

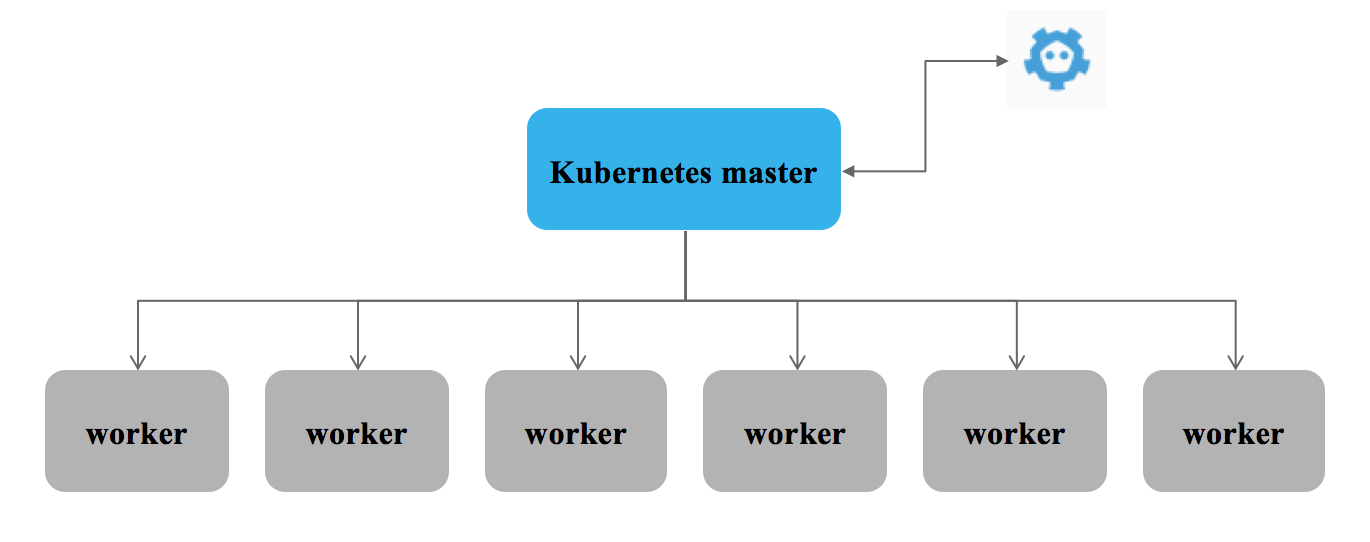
【摘要】Kubernetes是Google开源的Docker容器集群管理系统，为容器化的应用提供资源调度、部署运行、服务发现、扩容缩容等整一套功能。在应用部署这块，类似于其他系统，Kubernetes也提供了一些控制容器部署的方式，本文主要通过实践的方式介绍一下kubernetes的应用在部署时的部署策略。

先介绍几个概念性东西：

* **Pod**:是Kubernetes最基本的部署调度单元，可以包含container，逻辑上表示某种应用的一个实例。比如一个web站点应用由前端、后端及数据库构建而成，这三个组件将运行在各自的容器中，那么我们可以创建包含三个container的pod。
* **node**: 是 Kubernetes的worker节点，通常也称作为Minion node。除了运行一些kubernetes的组件以外（kubelet, kube-proxy等），还承担着运行容器服务的重任。
* **ReplicationController**：是pod的复制抽象，用于解决pod的扩容缩容问题。通常，分布式应用为了性能或高可用性的考虑，需要复制多份资源，并且根据负载情况动态伸缩。通过ReplicationController，我们可以指定一个应用需要几份复制，Kubernetes将为每份复制创建一个pod，并且保证实际运行pod数量总是与该复制数量相等(例如，当前某个pod宕机时，自动创建新的pod来替换)

环境介绍：

为了演示kubernetes应用部署策略，准备了7台机器（1个kubernetes master节点和6个kubernetes worker节点），如下图所示。



其中：

1) kubernetes master: hchenk8s1(ubuntu 16.04 LTS)

2) etcd: hchenk8s1（可以和kubernetes master不在一个节点上面）

3) worker nodes: hchenk8s2 - hchenk8s7（总共6台机器, 操作系统为ubuntu 16.04 LTS）

完了就开始环境搭建，这里就不演示了，环境搭建部分网上真的是一搜一堆的样子，可供参考的比较多。

下面直接进入主题。

目前，kubernetes提供了3中应用部署策略，下面一一进行介绍：

1. nodeSelector:

nodeSelector是kubernetes提供的最简单的一种应用部署策略，通过一种key=value的方式来部署用户的应用。

从这个参数就能看出来，这种策略的调度对象是node，也就是上面说的kubernetes的worker，说的更明白一点是，用户在创建应用的时候，可以通过nodeSelector来指定某个、或者某组具有某些属性的worker node来创建这些容器服务。这里既然提到了需要根据worker node的某些属性来创建这些容器服务，那就不得不介绍一下worker node的label.

Label: 标签的意思，使用在worker node上面顾名思义就是用来对worker node进行一些标记的。比如说worker node的cpu架构（ppc64, x86, etc）或者分组信息啊什么的。nodeSelector就是通过这些标签来选择应用到底要在哪些机器上去部署。

首先先查看当前kubernetes cluster的worker node的情况。

|  |
| --- |
| root@hchenk8s1:~# kubectl get nodes  NAME STATUS AGE  9.111.254.207 Ready,SchedulingDisabled 1d  9.111.254.208 Ready 1d  9.111.254.209 Ready 1d  9.111.254.212 Ready 1d  9.111.254.213 Ready 1d  9.111.254.214 Ready 1d  9.111.254.218 Ready 1d |

从输出可以看到目前测试集群中有6台worker node和一个不可调度的master节点。

下面我们通过nodeSelector来部署应用，并且应用需要部署在指定的机器上面。

在kubernetes集群中，kubelet会上报一些机器属性比如hostname, os, arch等信息记录在nodes的label里面。下面先查看一下这些label.

|  |
| --- |
| root@hchenk8s1:~# kubectl get nodes --show-labels  NAME STATUS AGE LABELS  9.111.254.207 Ready,SchedulingDisabled 1d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=9.111.254.207  9.111.254.208 Ready 1d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=9.111.254.208  9.111.254.209 Ready 1d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=9.111.254.209  9.111.254.212 Ready 1d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=9.111.254.212  9.111.254.213 Ready 1d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=9.111.254.213  9.111.254.214 Ready 1d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=9.111.254.214  9.111.254.218 Ready 1d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=9.111.254.218 |

从输出结果可以看到，每个node都有3个label分别是beta.kubernetes.io/arch，beta.kubernetes.io/os，kubernetes.io/hostname。下面通过hostname作为应用部署的选择策略来部署应用到9.111.254.218机器上面。

以nginx应用为例，准备一个容器应用部署的kubernetes的deployment文件。

|  |
| --- |
| ---  kind: Deployment  apiVersion: extensions/v1beta1  metadata:  name: nginx  spec:  replicas: 1  template:  metadata:  labels:  app: nginx  image: nginx\_1\_8\_1  spec:  hostNetwork: false  containers:  - name: nginx  image: nginx:1.8.1  imagePullPolicy: Always  ports:  - protocol: TCP  containerPort: 80  resources:  limits:  cpu: 1000m  memory: 1024Mi  **nodeSelector:**  **kubernetes.io/hostname: 9.111.254.218** |

在yaml文件中加入nodeSelector, 其中key和value分别为label的name和value.

下面就开始见证奇迹了。

通过kubectl创建应用容器服务。

|  |
| --- |
| root@hchenk8s1:~# kubectl create -f nginx.yaml  deployment "nginx" created  root@hchenk8s1:~# kubectl get deployment  NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE  nginx 1 1 1 1 9m  root@hchenk8s1:~# kubectl get pods -o wide  NAME READY STATUS RESTARTS AGE IP NODE  nginx-1245594662-sjjp9 1/1 Running 0 1m 10.1.20.130 9.111.254.218 |

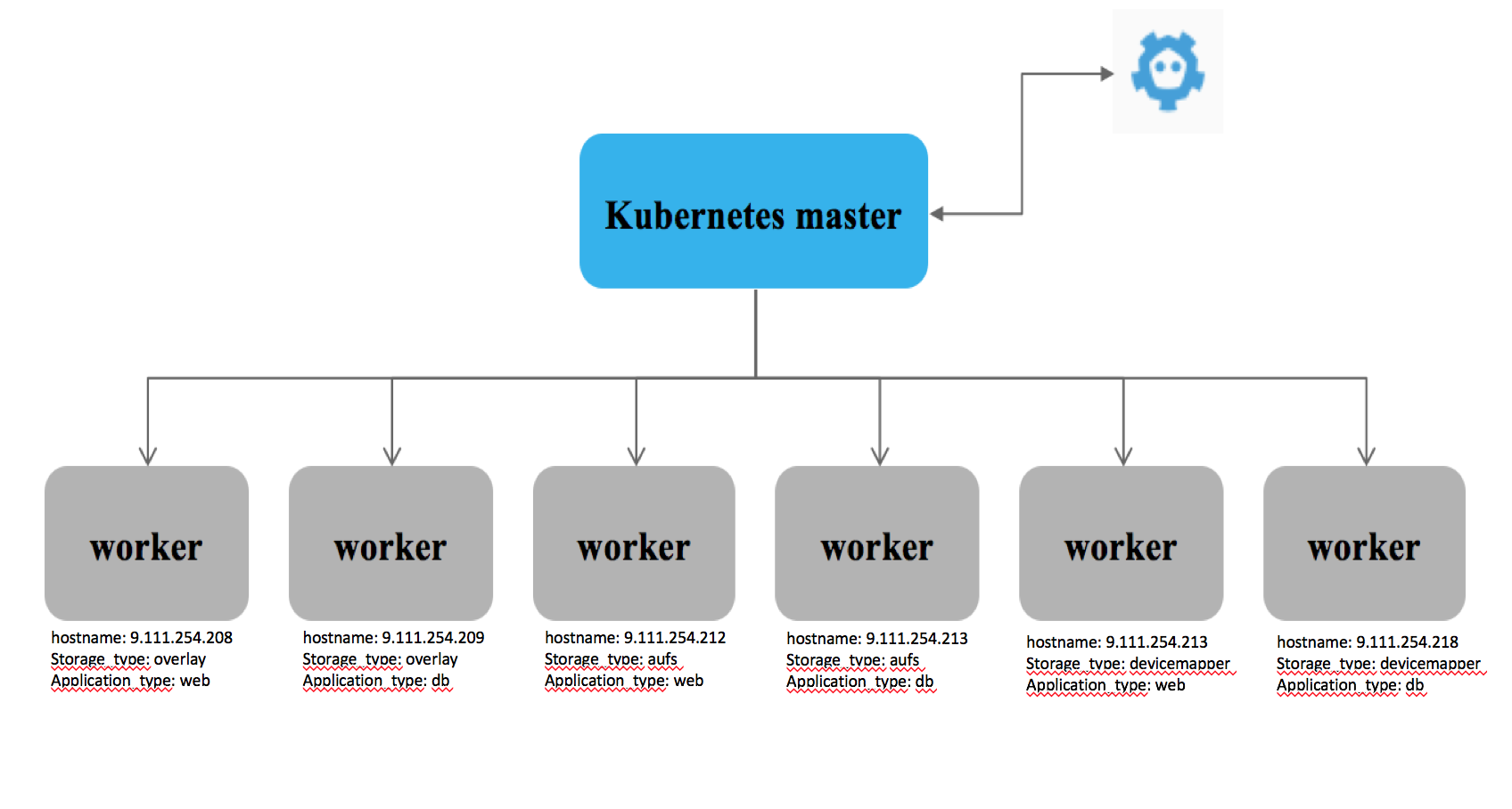
从输出可以看到, nginx的容器服务已经部署到了刚刚指定的机器上面。

当然，nodeSelector自身可以支持多个选择条件，当创建应用的时候，nodeSelector里面的条件都满足的机器会被选择出来用来部署pod.

为了测试nodeSelector多条件支持的测试，我们对6个worker分别进行标记：

|  |
| --- |
| root@hchenk8s1:~# kubectl label node 9.111.254.208 storage\_type=overlay application\_type=web  node "9.111.254.208" labeled  root@hchenk8s1:~# kubectl label node 9.111.254.209 storage\_type=overlay application\_type=db  node "9.111.254.209" labeled  root@hchenk8s1:~# kubectl label node 9.111.254.212 storage\_type=aufs application\_type=web  node "9.111.254.212" labeled  root@hchenk8s1:~# kubectl label node 9.111.254.213 storage\_type=aufs application\_type=db  node "9.111.254.213" labeled  root@hchenk8s1:~# kubectl label node 9.111.254.214 storage\_type=devicemapper application\_type=web  node "9.111.254.214" labeled  root@hchenk8s1:~# kubectl label node 9.111.254.218 storage\_type=devicemapper application\_type=db  node "9.111.254.218" labeled |

标记后， 集群结构如下图所示。



下面通过应用部署来测试nodeSelector的多条件选择：

场景1:

创建一个nginx web服务，选择worker node上面，storage\_type标记为啊aufs的节点：

期望结果：

nginx web服务部署在节点9.111.254.212上面

步骤：

准备需要创建服务所需要的yaml文件：

|  |
| --- |
| ---  kind: Deployment  apiVersion: extensions/v1beta1  metadata:  name: nginx  spec:  replicas: 1  template:  metadata:  labels:  app: nginx  image: nginx\_1\_8\_1  spec:  hostNetwork: false  containers:  - name: nginx  image: nginx:1.8.1  imagePullPolicy: IfNotPresent  ports:  - protocol: TCP  containerPort: 80  resources:  limits:  cpu: 1000m  memory: 1024Mi  **nodeSelector:**  **storage\_type: aufs**  **application\_type: web** |

从上面的yaml文件可以看到，nodeSelector里面定义了两个条件，分别是storage\_type和application\_type， 应用只有创建在两个条件同时满足的节点上面。

下面开始创建容器服务。

|  |
| --- |
| root@hchenk8s1:~# kubectl create -f nginx.yaml  deployment "nginx" created  root@hchenk8s1:~# kubectl get deployment  NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE  nginx 1 1 1 1 46s  root@hchenk8s1:~# kubectl get pods -o wide  NAME READY STATUS RESTARTS AGE IP NODE  nginx-2704164239-lr0gj 1/1 Running 0 1m 10.1.58.235 9.111.254.212 |

从输出可以看到，nginx服务已经选择机器9.111.254.212去部署应用。

场景2:

创建一个nginx web服务，选择worker node上面，storage\_type标记为啊btrfs的节点：

期望结果：

nginx web选择不到合适的机器部署应用。

步骤：

准备需要创建服务所需要的yaml文件：

|  |
| --- |
| ---  kind: Deployment  apiVersion: extensions/v1beta1  metadata:  name: nginx  spec:  replicas: 1  template:  metadata:  labels:  app: nginx  image: nginx\_1\_8\_1  spec:  hostNetwork: false  containers:  - name: nginx  image: nginx:1.8.1  imagePullPolicy: IfNotPresent  ports:  - protocol: TCP  containerPort: 80  resources:  limits:  cpu: 1000m  memory: 1024Mi  **nodeSelector:**  **storage\_type: btrfs**  **application\_type: web** |

下面开始创建容器服务。

|  |
| --- |
| root@hchenk8s1:~# kubectl create -f nginx.yaml  deployment "nginx" created  root@hchenk8s1:~# kubectl get deployment  NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE  nginx 1 1 1 0 58s  root@hchenk8s1:~# kubectl get pods -o wide  NAME READY STATUS RESTARTS AGE IP NODE  nginx-155862529-m5pr1 0/1 Pending 0 1m <none>  root@hchenk8s1:~# kubectl describe pods nginx-155862529-m5pr1  Name: nginx-155862529-m5pr1  Namespace: default  Node: /  Labels: app=nginx  pod-template-hash=155862529  Status: Pending  IP:  Controllers: ReplicaSet/nginx-155862529  Containers:  nginx:  Image: nginx:latest  Port: 80/TCP  Limits:  cpu: 1  memory: 1Gi  Requests:  cpu: 1  memory: 1Gi  Volume Mounts:  /var/run/secrets/kubernetes.io/serviceaccount from default-token-5rj87 (ro)  Environment Variables: <none>  Conditions:  Type Status  PodScheduled False  Volumes:  default-token-5rj87:  Type: Secret (a volume populated by a Secret)  SecretName: default-token-5rj87  QoS Class: Guaranteed  Tolerations: <none>  Events:  FirstSeen LastSeen Count From SubObjectPath Type Reason Message  --------- -------- ----- ---- ------------- -------- ------ -------  **1m 27s 8 {default-scheduler } Warning FailedScheduling pod (nginx-155862529-m5pr1) failed to fit in any node**  **fit failure summary on nodes : MatchNodeSelector (6)** |

从输出可以看到，pod创建失败了，原因是没有找到合适的机器去部署。

总结一下：nodeSelector通过label选择机制，提供了比较简单直观的pod部署策略，从一些方面实现了节点的亲和／反亲和的策略。虽然现在仍然存在在kubernetes中，不过相信这个功能会慢慢被接下来要提到的node Affinity和inter-pod affinity取而代之。

2. NodeAffinity

NodeAffinity是kubernetes 1.2的时候集成进来的，概念上类似于上面介绍的nodeSelector, 通过对node label的选择来部署你的pod的。

先说说nodeAffinity的类型：

目前nodeAffinity支持requiredDuringSchedulingIgnoredDuringExecution和preferredDuringSchedulingIgnoredDuringExecution这两种类型。从字面意思就能看到，类型一的要求要比类型二的苛刻的多，对于类型一来说，更像是上面介绍的nodeSelector的高级版，而对于类型二来说，在创建pod的时候会根据各种调度条件对可调度的机器进行排序，并且不会像类型一那样，因为资源不够或者一些其他原因而创建失败，退而求其次来去选择其他的机器继续创建。

在这两种类型中，ignoredDuringExecution的意思是在node在运行期间如果label发生了变化，之间通过这些类型部署的pod不会因为node label的变化而去重新部署来满足已经定义好的亲和／反亲和的策略。不过社区计划会针对这些case提供requiredDuringSchedulingRequiredDuringExecution的类型来应对因为node label变化，定义的亲和／反亲和的策略发生变化的问题，当然，pod可能就需要重新部署来适应已经发生的变化。

下面设计一个场景还试一下：

场景1:

集群中的6个worker node分别属于3个不同的组，这里分别命名为group1, group2, group3. 需要部署一个nignx应用，并且有4个副本，要求nignx应用部署在除了group3以外的其他group上面。

期望结果：

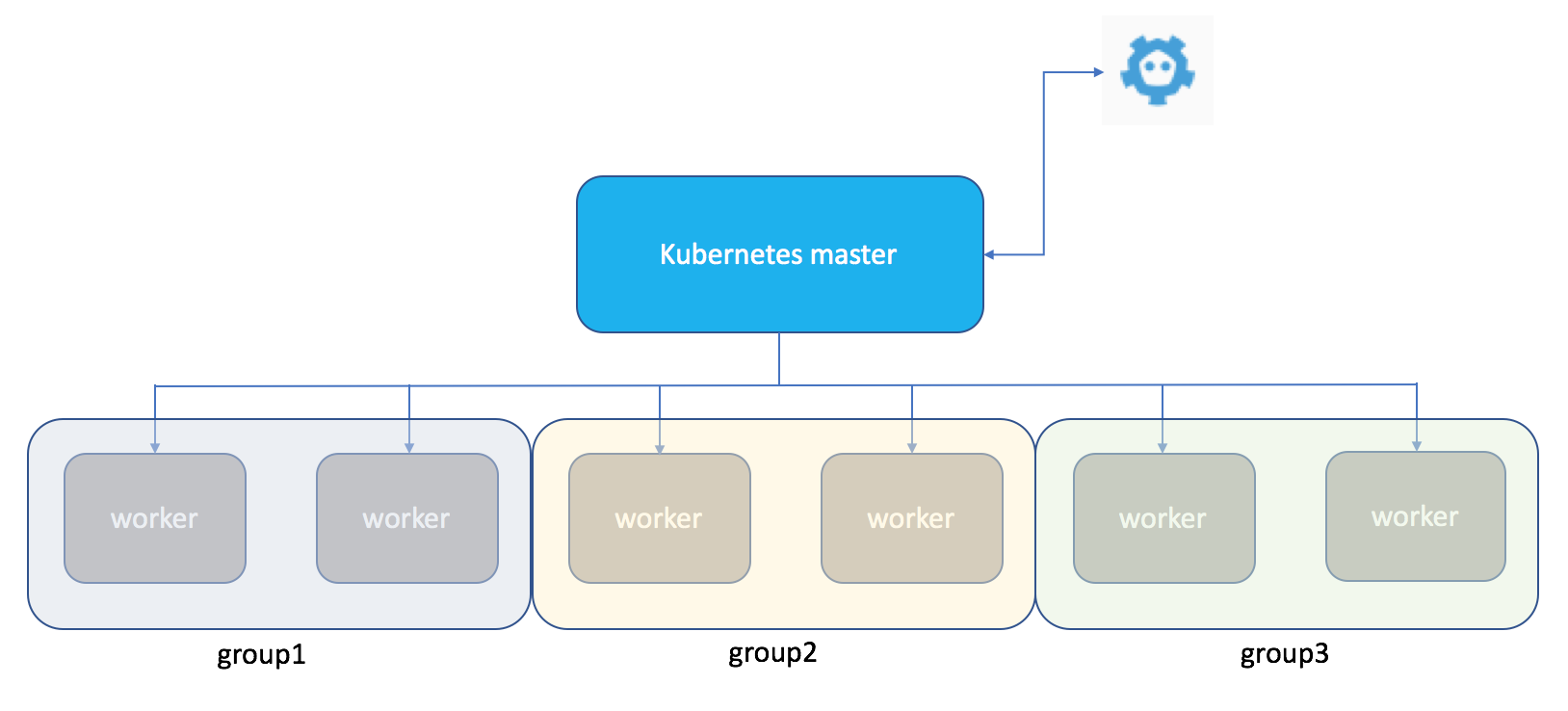
nginx的应用能部署在group1和group2里的worker node。

步骤：

首先，对集群中的worker node添加label来标识组信息。

|  |
| --- |
| root@hchenk8s1:~# kubectl label node 9.111.254.208 group=group1  node "9.111.254.208" labeled  root@hchenk8s1:~# kubectl label node 9.111.254.209 group=group1  node "9.111.254.209" labeled  root@hchenk8s1:~# kubectl label node 9.111.254.212 group=group2  node "9.111.254.212" labeled  root@hchenk8s1:~# kubectl label node 9.111.254.213 group=group2  node "9.111.254.213" labeled  root@hchenk8s1:~# kubectl label node 9.111.254.214 group=group3  node "9.111.254.214" labeled  root@hchenk8s1:~# kubectl label node 9.111.254.218 group=group3  node "9.111.254.218" labeled |

标识完后的worker node信息如下图所示。



下面准备一个用来测试的yaml文件：

|  |
| --- |
| ---  kind: Deployment  apiVersion: extensions/v1beta1  metadata:  name: nginx  spec:  replicas: 4  template:  metadata:  labels:  app: nginx  annotations:  **scheduler.alpha.kubernetes.io/affinity: >**  **{**  **"nodeAffinity": {**  **"requiredDuringSchedulingIgnoredDuringExecution": {**  **"nodeSelectorTerms": [**  **{**  **"matchExpressions": [**  **{**  **"key": "group",**  **"operator": "In",**  **"values": ["group1", "group2"]**  **}**  **]**  **}**  **]**  **}**  **}**  **}**  spec:  hostNetwork: false  containers:  - name: nginx  image: nginx:latest  imagePullPolicy: IfNotPresent  ports:  - protocol: TCP  containerPort: 80  resources:  limits:  cpu: 500m  memory: 512Mi |

下面开始创建容器服务。

|  |
| --- |
| root@hchenk8s1:~# kc create -f nginx\_nodeaffinity.yaml  deployment "nginx" created  root@hchenk8s1:~# kc get pods -o wide  NAME READY STATUS RESTARTS AGE IP NODE  nginx-3792017226-5cn0s 1/1 Running 0 1m 10.1.36.194 9.111.254.213  nginx-3792017226-ljq9h 1/1 Running 0 1m 10.1.56.66 9.111.254.208  nginx-3792017226-qfbvs 1/1 Running 0 1m 10.1.64.66 9.111.254.212  nginx-3792017226-tdm23 1/1 Running 0 1m 10.1.183.3 9.111.254.209 |

从测试结果可以看到，4个副本分别部署在了group1和group2上的机器。

上面的例子中用了In的operator，matchExpressions的operator大概有一下几种：

In: 凡是满足values里面条件的机器都会被选择出来。以上面的例子为例，凡是满足group=group1或者group=group2的机器都会被选择出来。

NotIn: 和In相反，凡是满足values里面条件的机器都会被剔除出去。如果以上面的例子为例，operator换成NotIn, 那么group=group1以及group=group2的机器就会被剔除出去，而group=group3的机器则会被选择出来。

Exists: 和In比较类似，凡是有某个标签的机器都会被选择出来。使用Exists的operator的话，values里面就不能写东西了。

DoesNotExist: 和Exists相反，凡是不具备某个标签的机器则会被选择出来。和Exists的Operator一样，values里面也不能写东西了。

Gt: greater than的意思，表示凡是某个value大于设定的值的机器则会被选择出来。

Lt: less than的意思，表示凡是某个value小于设定的值的机器则会被选择出来。

下面2个例子分别看看其他几个operator的使用以及测试结果。

场景2:

集群中的6个worker node，其中的2台标记了network的标签，而其他的4台没有network标签。通过deployment创建一个nginx应用，并且nginx应用有4个副本，通过nodeAffinity选择有network标签的机器进行应用部署。

期望结果:

nginx的应用能部署在有network的标签的机器上面。

步骤:

首先，对集群中的worker node添加label来标识组信息，通过命令可以查看当前集群中的worker node的label信息。

|  |
| --- |
| root@hchenk8s1:~# kubectl get nodes --show-labels  NAME STATUS AGE LABELS  9.111.254.208 Ready 6d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,group=group1,kubernetes.io/hostname=9.111.254.208,**network=calico**  9.111.254.209 Ready 6d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,group=group1,kubernetes.io/hostname=9.111.254.209,**network=calico**  9.111.254.212 Ready 6d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,group=group2,kubernetes.io/hostname=9.111.254.212  9.111.254.213 Ready 6d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,group=group2,kubernetes.io/hostname=9.111.254.213  9.111.254.214 Ready 6d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,group=group3,kubernetes.io/hostname=9.111.254.214  9.111.254.218 Ready 6d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,group=group3,kubernetes.io/hostname=9.111.254.218 |

准备创建nginx应用的deployment

|  |
| --- |
| ---  kind: Deployment  apiVersion: extensions/v1beta1  metadata:  name: nginx  spec:  replicas: 4  template:  metadata:  labels:  app: nginx  annotations:  scheduler.alpha.kubernetes.io/affinity: >  {  "nodeAffinity": {  "requiredDuringSchedulingIgnoredDuringExecution": {  "nodeSelectorTerms": [  {  "matchExpressions": [  **{**  **"key": "network",**  **"operator": "Exists"**  **}**  ]  }  ]  }  }  }  spec:  hostNetwork: false  containers:  - name: nginx  image: nginx:latest  imagePullPolicy: IfNotPresent  ports:  - protocol: TCP  containerPort: 80  resources:  limits:  cpu: 500m  memory: 512Mi |

从上面的yaml文件可以看到，matchExpression里面定义了nodeAffinity的选择条件，从上面的例子可以看到，nginx应用期望能创建在有network label的机器上。

下面开始创建应用

|  |
| --- |
| root@hchenk8s1:~# kubectl create -f nginx\_exist.yaml  deployment "nginx" created  root@hchenk8s1:~# kubectl get pods -o wide  NAME READY STATUS RESTARTS AGE IP NODE  nginx-3031338627-7bbfc 1/1 Running 0 7s 10.1.183.17 9.111.254.209  nginx-3031338627-cd1jz 1/1 Running 0 7s 10.1.56.80 9.111.254.208  nginx-3031338627-wslpb 1/1 Running 0 7s 10.1.183.16 9.111.254.209  nginx-3031338627-zgrxn 1/1 Running 0 7s 10.1.56.79 9.111.254.208 |

从测试结果可以看到，4个副本分别部署在了有network label的机器上面。

场景3:

集群中的6个worker node，其中的2台有kernel-version标签，用来记录机器的内核版本。通过deployment创建一个nginx应用，并且nginx应用有4个副本，通过nodeAffinity选择内核版本范围来进行应用部署。

期望结果：

期望nginx的应用部署在kerver-version大于0320的机器上面。

步骤：

首先为集群中的机器添加lable来标示机器的内核版本信息：

|  |
| --- |
| root@hchenk8s1:~# kc get nodes --show-labels -l worker=true  NAME STATUS AGE LABELS  9.111.254.208 Ready 6d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,group=group1,**kernel-version=0310**,kubernetes.io/hostname=9.111.254.208,worker=true  9.111.254.209 Ready 6d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,group=group1,**kernel-version=0404**,kubernetes.io/hostname=9.111.254.209,worker=true  9.111.254.212 Ready 6d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,group=group2,kubernetes.io/hostname=9.111.254.212,worker=true  9.111.254.213 Ready 6d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,group=group2,kubernetes.io/hostname=9.111.254.213,worker=true  9.111.254.214 Ready 6d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,group=group3,kubernetes.io/hostname=9.111.254.214,worker=true  9.111.254.218 Ready 6d beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,group=group3,kubernetes.io/hostname=9.111.254.218,worker=true |

从上面的输出可以看到，9.111.254.208的kernel-version为0310，9.111.254.209的kernel-version为0404.

下面准备一个nginx的yaml文件，用来创建nginx服务：

|  |
| --- |
| ---  kind: Deployment  apiVersion: extensions/v1beta1  metadata:  name: nginx  spec:  replicas: 4  template:  metadata:  labels:  app: nginx  annotations:  scheduler.alpha.kubernetes.io/affinity: >  {  "nodeAffinity": {  "requiredDuringSchedulingIgnoredDuringExecution": {  "nodeSelectorTerms": [  {  **"matchExpressions": [**  **{**  **"key": "kernel-version",**  **"operator": "Gt",**  **"values": ["0320"]**  **}**  ]  }  ]  }  }  }  spec:  hostNetwork: false  containers:  - name: nginx  image: nginx:latest  imagePullPolicy: IfNotPresent  ports:  - protocol: TCP  containerPort: 80  resources:  limits:  cpu: 200m  memory: 256Mi |

上面的文件表示的策略是期望创建服务到kernel-version大于0320的机器上面。

下面开始创建；

|  |
| --- |
| root@hchenk8s1:~# kubectl create -f nginx\_gt.yaml  deployment "nginx" created  root@hchenk8s1:~# kubectl get pods -o wide  NAME READY STATUS RESTARTS AGE IP NODE  nginx-4087060041-2x9lw 1/1 Running 0 4s 10.1.183.26 9.111.254.209  nginx-4087060041-4x1dd 1/1 Running 0 4s 10.1.183.23 9.111.254.209  nginx-4087060041-bgt0z 1/1 Running 0 4s 10.1.183.24 9.111.254.209  nginx-4087060041-brgb3 1/1 Running 0 4s 10.1.183.25 9.111.254.209 |

从测试结果可以看到，4个副本都创建在了kernel-version为0404的机器上。

3. Inter-pod affinity/anti-affinity

Inter-pod affinity/anti-affinity是kubernetes 1.4开始支持的，pod的部署策略不再单单的只是通过node label的选择，而是可以从pod层面通过pod的 label来部署自己的应用，简单点说就是你可以通过inter-pod affinity/anti-affinity来决定自己的容器应用亲近或者远离具有某些label的容器应用。

和nodeAffinity一样，inter-pod affinity/anti-affinity也有requiredDuringSchedulingIgnoredDuringExecution和preferredDuringSchedulingIgnoredDuringExecution两种类型，分别表示"hard"和"soft"两种需求。

场景1.

集群中有个db容器服务(mysql)，通过使用inter-pod affinity使得数据库上层服务(wordpress)能够和db容器服务在一台机器上。

步骤：

1. 首先准备mysql.yaml文件并且创建容器服务。

|  |
| --- |
| ---  apiVersion: v1  kind: Pod  metadata:  name: mysql  labels:  name: mysql  spec:  containers:  - resources:  limits :  cpu: 0.5  image: mysql:5.6  name: mysql  args:  - "--ignore-db-dir"  - "lost+found"  env:  - name: MYSQL\_ROOT\_PASSWORD  # change this  value: changeit  ports:  - containerPort: 3306  name: mysql |

下面开始创建：

|  |
| --- |
| root@hchenk8s1:~# kubectl create -f mysql.yaml  pod "mysql" created  root@hchenk8s1:~# kc get pods -owide --show-labels  NAME READY STATUS RESTARTS AGE IP NODE LABELS  mysql 1/1 Running 0 8m 10.1.226.66 9.111.254.214 name=mysql |

mysql创建好了，下面开始准备wordpress的yaml文件。

|  |
| --- |
| ---  apiVersion: v1  kind: Pod  metadata:  name: wordpress  spec:  **affinity:**  **podAffinity:**  **requiredDuringSchedulingIgnoredDuringExecution:**  **- labelSelector:**  **matchExpressions:**  **- key: name**  **operator: In**  **values:**  **- mysql**  **topologyKey: kubernetes.io/hostname**  containers:  - image: wordpress:4.7.3-apache  name: wordpress  env:  - name: WORDPRESS\_DB\_HOST  value: 10.1.226.66  - name: WORDPRESS\_DB\_PASSWORD  value: changeit  ports:  - containerPort: 80  name: wordpress |

yaml文件里面定义了podAffinity, matchExpressions里面指定了需要选择name <in> mysql的pod label来部署wordpress的pod.

下面开始创建:

|  |
| --- |
| root@hchenk8s1:~# kubectl create -f wordpress.yaml  pod "wordpress" created  root@hchenk8s1:~# kubectl get pods -owide --show-labels  NAME READY STATUS RESTARTS AGE IP NODE LABELS  mysql 1/1 Running 0 1d 10.1.226.66 9.111.254.214 name=mysql  wordpress 1/1 Running 0 1d 10.1.226.67 9.111.254.214 <none> |

从输出可以看到，wordpress和mysql的pod部署在了一台机器上面，实现了kubernetes.io/hostname上的亲和策略。

场景2.

集群中有个db容器服务(mysql)，通过使用inter-pod anti-affinity使得数据库上层服务(wordpress)能够和db容器服务不在一台机器上。

延用上面的mysql的pod, 直接从wordpress的应用开始。

首先还是从yaml文件的准备开始：

|  |
| --- |
| ---  apiVersion: v1  kind: Pod  metadata:  name: wordpress  spec:  **affinity:**  **podAntiAffinity:**  **requiredDuringSchedulingIgnoredDuringExecution:**  **- labelSelector:**  **matchExpressions:**  **- key: name**  **operator: In**  **values:**  **- mysql**  **topologyKey: kubernetes.io/hostname**  containers:  - image: wordpress:4.7.3-apache  name: wordpress  env:  - name: WORDPRESS\_DB\_HOST  value: 10.1.226.66  - name: WORDPRESS\_DB\_PASSWORD  value: changeit  ports:  - containerPort: 80  name: wordpress |

下面开始创建：

|  |
| --- |
| root@hchenk8s1:~# kubectl create -f wordpress.yaml  pod "wordpress" created  root@hchenk8s1:~# kubectl get pods -owide --show-labels  NAME READY STATUS RESTARTS AGE IP NODE LABELS  mysql 1/1 Running 0 1d 10.1.226.66 9.111.254.214 name=mysql  wordpress 1/1 Running 0 4m 10.1.36.207 9.111.254.213 <none> |

从测试结果可以看到，pod分布在了不同的机器上面。

另外，kubernetes的taints和tolerations也是用来对pod进行调度的，这个会在下期进行介绍。

作者简介：陈晖，供职于西安IBM, 主要从事云计算方面工作，涉及Openstack, Mesos, kubernetes等开源项目，在资源管理 ，资源调度等方面有比较丰富的经验 。