kubernetes相关概念

笔记本： 新课程笔记
创建时间： 2018/8/23 星期四 下午 10:00 **更新时间：** 2018/10/10 星期三 下午 10:06
作者： 306798658@qq.com

kubernetes内部组件工作原理 http://dockone.io/article/5108

Master

Master是整个集群的控制中心，kubernetes的所有控制指令都是发给master，它负责具体的执行过程。一般我们会把master独立于一台物理机或者一台虚拟机，它的重要性不言而喻。

master上有这些关键的进程：
Kubernetes API Server（kube-apiserver)，提供了HTTP Rest接口关键服务进程，是所有资源增、删、改、查等操作的唯一入口，也是集群控制的入口进程。
Kubernetes Controller Manager(kube-controlker-manager)，是所有资源对象的自动化控制中心，可以理解为资源对象的大总管。
Kubernetes Scheduler(kube-scheduler)，负责资源调度（pod调度）的进程，相当于公交公司的“调度室”。
etcd Server，kubernetes里所有资源对象的数据都是存储在etcd中的。

Node

除了Master，Kubernetes集群中其他机器被称为Node，早期版本叫做Minion。Node可以是物理机也可以是虚拟机，每个Node上会被分配一些工作负载（即，docker容器），当Node宕机后，其上面跑的应用会被转移到其他Node上。

Node上有这些关键进程：
kubelet：负责Pod对应容器的创建、启停等任务，同时与Master节点密切协作，实现集群管理的基本功能。
kube-proxy：实现Kubernetes Service的通信与负载均衡机制的重要组件。
Docker Engine（docker）：Docker引擎，负责本机容器的创建和管理。

kubectl get nodes #查看集群中有多少个node
kubectl describe node <node name> #查看Node的详细信息

Pod

查看pod命令： kubectl get pods
查看容器命令： docker ps
可以看到容器和pod是有对应关系的，在我们做过的实验中，每个pod对应两个容器，一个是Pause容器，一个是rc里面定义的容器（实际上，每个pod里可以有多个应用容器）。这个Pause容器叫做“根容器”，只有当Pause容器“死亡”才会认为该pod“死亡”。Pause容器的IP以及其挂载的Volume资源会共享给该pod下的其他容器。

pod定义示例：

```
apiVersion: v1
kind: pod
metadata:
  name: myweb
  labels:
    name: myweb
spec:
  containers:
  - name: myweb
    image: kubeguide/tomcat-app:v1
    ports:
    - containerPort: 8080
    env:
    - name: MYSQL_SERVICE_HOST
      value: 'mysql'
    - name: MYSQL_SERVICE_PORT
```

value: '3306'

每个pod都可以对其能使用的服务器上的硬件资源进行限制（CPU、内存）。CPU限定的最小单位是1/1000个cpu，用m表示，如100m，就是0.1个cpu。内存限定的最小单位是字节，可以用Mi（兆） 表示，如128Mi就是128M。
在kubernetes里，一个计算资源进行配额限定需要设定两个参数：
1）requests：该资源的最小申请量
2）Limits：该资源允许的最大使用量。

资源限定示例：

```
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Label

Label是一个键值对，其中键和值都由用户自定义，Label可以附加在各种资源对象上，如Node、Pod、Service、RC等。一个资源对象可以定义多个Label，同一个Label也可以被添加到任意数量的资源对象上。Label可以在定义对象时定义，也可以在对象创建完后动态添加或删除。

Label示例：
"release":"stable", "environment":"dev", "tier":"backend"等等。

RC

RC是kubernetes中核心概念之一，简单说它定义了一个期望的场景，即声明某种pod的副本数量在任意时刻都符合某个预期值，RC定义了如下几个部分：
1）pod期待的副本数
2）用于筛选目标pod的Label Selector
3）创建pod副本的模板（template）

RC一旦被提交到kubernetes集群后，Master节点上的Controller Manager组件就会接收到该通知，它会定期巡检集群中存活的pod，并确保pod数量符合RC的定义值。可以说通过RC，kubernetes实现了用户应用集群的高可用性，并且大大减少了管理员在传统IT环境中不得不做的诸多手工运维工作，比如编写主机监控脚本、应用监控脚本、故障恢复处理脚本等

RC工作流程（假如，集群中有3个Node）：
1）RC定义2个pod副本
2）假设系统会在2个Node上（Node1和Node2）创建pod
3）如果Node2上的pod（pod2）意外终止，这很有可能是因为Node2宕机
4）则会创建一个新的pod，假设会在Node3上创建pod3，当然也有可能在Node1上创建pod3

RC中动态修改pod副本数量：
kubectl scale rc <rc name> --replicas=n

利用动态修改pod的副本数，可以实现应用的动态升级（滚动升级）：
1）以新版本的镜像定义新的RC，但pod要和旧版本保持一致（由Label决定）
2）新版本每增加1个pod，旧版本就减少一个pod，始终保持固定的值
3）最终旧版本pod数为0，全部为新版本

删除RC
kubectl delete rc <rc name>
删除RC后，RC对应的pod也会被删除掉

Deployment

在1.2版本引入的概念，目的是为了解决pod编排问题，在内部使用了Replica Set，它和RC比较，相似度为90%以上，可以认为是RC的升级版。 跟RC比较，最大的一个特点是可以知道pod部署的进度。

Deployment示例：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      containers:
      - name: tomcat-demo
        image: tomcat
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
```

```
kubectl create  -f tomcat-deployment.yaml
kubectl get deployment
```

HPA(Horizontail Pod Autoscaler)

在1.1版本，kubernetes官方发布了HPA，实现pod的动态扩容、缩容，它属于一种kubernetes的资源对象。它通过追踪分析RC控制的所有目标pod的负载变化情况，来决定是否需要针对性地调整目标Pod的副本数，这是HPA的实现原理。

pod负载度量指标：

1）CpuUtilizationPercentage

目标pod所有副本自身的cpu利用率平用均值。一个pod自身的cpu利用率 = 该pod当前cpu的使用量 / pod Request值。如果某一个时刻，CPUUtilizationPercentage的值超过了80%，则判定当前的pod已经不够支撑业务，需要增加pod。

2）应用程序自定义的度量指标，比如服务每秒内的请求数（TPS或QPS）

HPA示例：

```
apiVerion: autosacling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    kind: Deployment
    name: php-apache
  targetCPUUtilizationPercentage: 90
```

说明：HPA控制的目标对象是一个名叫php-apache的Deployment里的pod副本，当cpu平均值超过90%时就会扩容，pod副本数控制范围是1 - 10.

除了以上的xml文件定义HPA外，也可以用命令行的方式来定义：
kubectl autoscale deployment php-apache --cpu-percent=90 --min=1 --max=10

Service

Service是kubernetes中最核心的资源对象之一，Service可以理解成是微服务架构中的一个“微服务”，pod、RC、Deployment都是为Service提供嫁衣的。

简单讲一个service本质上是一组pod组成的一个集群，前面我们说过service和pod之间是通过Label来串起来的，相同Service的pod的Label一样。同一个service下的所有pod是通过kube-proxy实现负载均衡，而每个service都会分配一个全局唯一的虚拟ip，也叫做cluster ip。在该service整个生命周期内，cluster ip是不会改变的，而在kubernetes中还有一个dns服务，它把service的name和cluster ip映射起来。

service示例:(文件名tomcat-service.yaml)

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
  ports:
    - port: 8080
    selector:
      tier: frontend
```

```
kubectl create -f tomcat-service.yaml
kubectl get endpoints  //查看pod的IP地址以及端口
kubectl get svc tomcat-service -o yaml  //查看service分配的cluster ip
```

多端口的service

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
  ports:
    - port: 8080
      name: service-port
    - port: 8005
      name: shutdown-port
    selector:
      tier: frontend
```

对于cluster ip有如下限制：
1）Cluster ip无法被ping通，因为没有实体网络来响应
2）Cluster ip和Service port组成了一个具体的通信端口，单独的Cluster ip不具备TCP/IP通信基础，它们属于一个封闭的空间。
3）在kubernetes集群中，Node ip、pod ip、cluster ip之间的通信，采用的是kubernetes自己设计的一套编程方式的特殊路由规则。

要想直接和service通信，需要一个Nodeport，在service的yaml文件中定义：

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
  ports:
```

```
  - port: 8080
    nodeport: 31002
  selector:
    tier: frontend
```

它实质上是把cluster ip的port映射到了node ip的nodeport上了

Volume(存储卷)
Volume是pod中能够被多个容器访问的共享目录，kubernetes中的volume和docker中的volume不一样，主要有以下几个方面：
1）kubernetes的volume定义在pod上，然后被一个pod里的多个容器挂载到具体的目录下
2）kubernetes的volume与pod生命周期相同，但与容器的生命周期没关系，当容器终止或者重启时，volume中的数据并不会丢失
3）kubernetes支持多种类型的volume，如glusterfs，ceph等先进的分布式文件系统

如何定义并使用volume呢？只需要在定义pod的yaml配置文件中指定volume相关配置即可：

```
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      volumes:
        - name: datavol
          emptyDir: {}
      containers:
      - name: tomcat-demo
        image: tomcat
        volumeMounts:
          - mountPath: /mydata-data
            name: datavol
        imagePullPolicy: IfNotPresent
```

说明: volume名字是datavol，类型是emptyDir，将volume挂载到容器的/mydata-data目录下

volume的类型：
1）emptyDir
是在pod分配到node时创建的，初始内容为空，不需要关心它将会在宿主机（node）上的哪个目录下，因为这是kubernetes自动分配的一个目录，当pod从node上移除，emptyDir上的数据也会消失。所以，这种类型的volume不适合存储永久数据，适合存放临时文件。

2）hostPath
hostPath指定宿主机（node）上的目录路径，然后pod里的容器挂载该共享目录。这样有一个问题，如果是多个node，虽然目录一样，但是数据不能做到一致，所以这个类型适合一个node的情况。

配置示例：

```
      volumes:
        - name: "persistent-storage"
          hostPath:
            path: "/data"
```

3）gcePersistentDisk
使用Google公有云GCE提供的永久磁盘（PD）存储volume数据。毫无疑问，使用gcePersistentDisk的前提是kubernetes的node是基于GCE的。

配置示例：

```
      volumes:
```

```
      - name: test-volume
        gcePersistentDisk:
          pdName: my-data-disk
          fsType: ext4
```

4）awsElasticBlockStore
与GCE类似，该类型使用亚马逊公有云提供的EBS Volume存储数据，使用它的前提是Node必须是aws EC2。

5）NFS
使用NFS作为volume载体。

示例：

```
    volumes:
      - name: "NFS"
        NFS:
          server: ip地址
          path: "/"
```

6）其他类型
iscsi
flocker
glusterfs
rbd
gitRepo: 从git仓库clone一个git项目，以供pod使用
secret: 用于为pod提供加密的信息

persistent volume（PV）
PV可以理解成kubernetes集群中某个网络存储中对应的一块存储，它与volume类似，但有如下区别：
1）PV只能是网络存储，不属于任何Node，但可以在每个Node上访问到
2）PV并不是定义在pod上，而是独立于pod之外定义
3）PV目前只有几种类型：GCE Persistent Disk、NFS、RBD、iSCSCI、AWS ElasticBlockStore、GlusterFS

如下是NFS类型的PV定义：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  nfs:
    path: /somepath
    server: ip
```

其中accessModes是一个重要的属性，目前有以下类型：
ReadWriteOnce: 读写权限，并且只能被单个Node挂载
ReadOnlyMany: 只读权限，允许被多个Node挂载
ReadWriteMany：读写权限，允许被多个Node挂载

如果某个pod想申请某种条件的PV，首先需要定义一个PersistentVolumeClaim（PVC）对象：

```
kind: persistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
```

```
    requests:
      storage: 8Gi
```

然后在pod的vomume定义中引用上面的PVC：

```
volumes:
  - name: mypd
    persistentVolumeClaim:
      ClaimName: myclaim
```

Namespace（命名空间）
当kubernetes集群中存在多租户的情况下，就需要有一种机制实现每个租户的资源隔离。而namespace的目的就是为了实现资源隔离。

kubectl get namespace //查看集群所有的namespace

定义namespace：

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev
```

kubectl create -f dev-namespace.yaml //创建dev namespace

然后再定义pod，指定namespace

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: dev
spec:
  containers:
  - image: busybox
    command:
      - sleep
      - "500"
    name: busybox
```

查看某个namespace下的pod：
kubectl get pod --namespace=dev