openshift yaml文件解读

v1.0

2017.07.20

文档变更历史清单

|  |  |  |  |
| --- | --- | --- | --- |
| 文件版本号 | 修正日期 | 修正人 | 备 注 |
| 1.0 | 2017-07-20 | 蔡峰 | 创建文档 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目录

[yaml文件解读 4](#_Toc488332675)

[1.1. 一些通用注意事项 4](#_Toc488332676)

[1.2. Replicationcontrollers(rc) 4](#_Toc488332677)

[1.2.1. 范例1（简单tomcat） 4](#_Toc488332678)

[1.2.2. 范例2（支持格式大全） 5](#_Toc488332679)

[1.2.3. 范例3（较全） 11](#_Toc488332680)

[1.3. pod的liveness & readiness 16](#_Toc488332681)

[1.3.1. exec 17](#_Toc488332682)

[1.3.2. httpget 18](#_Toc488332683)

[1.3.3. 更多参数配置 18](#_Toc488332684)

[1.4. ReplicaSet（rs） 19](#_Toc488332685)

[1.5. Deployment（deploy） 20](#_Toc488332686)

[1.6. Pod 21](#_Toc488332687)

[1.6.1. ubuntu 21](#_Toc488332688)

[1.6.2. nginx 22](#_Toc488332689)

[1.6.3. pod中多容器 22](#_Toc488332690)

[1.7. job 22](#_Toc488332691)

[1.8. template 24](#_Toc488332692)

[1.9. Service(svc) 27](#_Toc488332693)

[1.9.1. ClusterIP 27](#_Toc488332694)

[1.9.2. nodeport 28](#_Toc488332695)

[1.9.3. loadBalancer 29](#_Toc488332696)

[1.10. Persistentvolumes(pv) 29](#_Toc488332697)

[1.10.1. 访问模式 29](#_Toc488332698)

[1.10.2. 回收策略 30](#_Toc488332699)

[1.10.3. 卷的状态 30](#_Toc488332700)

[1.10.4. nfs 范例 31](#_Toc488332701)

[1.10.5. hostpath范例 31](#_Toc488332702)

[1.11. Persistentvolumeclaims(pvc) 32](#_Toc488332703)

[1.11.1. 范例1 32](#_Toc488332704)

[1.11.2. 范例2 32](#_Toc488332705)

[1.12. 对接glusterfs 33](#_Toc488332706)

[1.13. Horizontal Pod Autoscaling（hpa） 36](#_Toc488332707)

[1.14. configmap（cm） 37](#_Toc488332708)

[1.14.1. 创建ConfigMaps 38](#_Toc488332709)

[1.14.2. Pod中使用ConfigMap 41](#_Toc488332710)

[1.15. 多资源共存的yaml文件书写规则 45](#_Toc488332711)

[1.16. 参考文档 46](#_Toc488332712)

# yaml文件解读

## 一些通用注意事项

ports，volumeMounts，volumes、env、securityContext等，有多个子项的时候。以上参数只能出现一次。

|  |
| --- |
| 错误示例：  volumeMounts:  - mountPath: /home/netbank/share  name: share  ~~volumeMounts:~~ 此行多余，将会导致上面的Volumemounts参数不生效  - mountPath: /home/netbank/facenas  name: facenas |

注意缩进，特别是容器中的参数，多和范本中对比，确认是和image平级或者是和containers平级。

yaml文件中，一些非全局的资源，比如rc，svc，pvc。最好在文件定义好namespace。防止在执行命令oc create -f的时候所在的project不对，导致一些异常。

同一应用的资源，pv，pvc，rc，svc最好带上相同的label，便于查找和删除。

## Replicationcontrollers(rc)

ReplicationCtronller用来确保容器应用的副本数始终保持在用户定义的副本数，即如果有容器异常退出，会自动创建新的Pod来替代；而如果异常多出来的容器也会自动回收。

### 范例1（简单tomcat）

|  |
| --- |
| apiVersion: v1  kind: ReplicationController  metadata:  name: tomcat1 #rc资源的名称  namespace: default #命令空间，也是openshift里的project  labels:  app: tomcat1 #标签  spec:  replicas: 2 #pod实例数  selector:  app: tomcat1  template:  metadata:  name: tomcat1  labels:  app: tomcat1  spec:  containers:  - image: tomcat:latest #镜像名称  name: tomcat1  ports:  - containerPort: 8080 #容器内部端口  name: app-port  protocol: TCP |
|  |

spec.replicas：副本数量2

spec.selector：RC通过spec.selector来筛选要控制的Pod

spec.template：这里写Pod的定义（但不需要apiVersion和kind）

spec.template.metadata.labels：Pod的label，可以看到这个label与spec.selector相同

### 范例2（支持格式大全）

|  |
| --- |
| kind: '' //必选，ReplicationController  apiVersion: '' //必选，v1  metadata: //必选，元数据  name: '' //必选，rc名称  generateName: ''  namespace: '' //必选，命名空间，默认default  selfLink: ''  uid: ''  resourceVersion: ''  generation: {}  creationTimestamp: ''  deletionTimestamp: ''  deletionGracePeriodSeconds: {}  labels: any  annotations: any  spec: //必选，详细描述  replicas: {} //必选，pod副本数，设置0表示不创建pod  selector: any //必选，rc label选择器  template: //必选，容器的定义  metadata:  name: ''  generateName: ''  namespace: ''  selfLink: ''  uid: ''  resourceVersion: ''  generation: {}  creationTimestamp: ''  deletionTimestamp: ''  deletionGracePeriodSeconds: {}  labels: any  annotations: any  spec:  volumes:  - name: ''  hostPath:  path: ''  emptyDir:  medium: ''  gcePersistentDisk:  pdName: ''  fsType: ''  partition: {}  readOnly: false  awsElasticBlockStore:  volumeID: ''  fsType: ''  partition: {}  readOnly: false  gitRepo:  repository: ''  revision: ''  secret:  secretName: ''  nfs:  server: ''  path: ''  readOnly: false  iscsi:  targetPortal: ''  iqn: ''  lun: {}  fsType: ''  readOnly: false  glusterfs:  endpoints: ''  path: ''  readOnly: false  persistentVolumeClaim:  claimName: ''  readOnly: false  rbd:  monitors:  - ''  image: ''  fsType: ''  pool: ''  user: ''  keyring: ''  secretRef:  name: ''  readOnly: false  cinder:  volumeID: ''  fsType: ''  readOnly: false  cephfs:  monitors:  - ''  user: ''  secretFile: ''  secretRef:  name: ''  readOnly: false  flocker:  datasetName: ''  downwardAPI:  items:  - path: ''  fieldRef:  apiVersion: ''  fieldPath: ''  fc:  targetWWNs:  - ''  lun: {}  fsType: ''  readOnly: false  metadata:  items:  - name: ''  fieldRef:  apiVersion: ''  fieldPath: ''  containers:  - name: ''  image: ''  command:  - ''  args:  - ''  workingDir: ''  ports:  - name: ''  hostPort: {}  containerPort: {}  protocol: ''  hostIP: ''  env:  - name: ''  value: ''  valueFrom:  fieldRef:  apiVersion: ''  fieldPath: ''  resources:  limits: any  requests: any  volumeMounts:  - name: ''  readOnly: false  mountPath: ''  livenessProbe:  exec:  command:  - ''  httpGet:  path: ''  port: ''  host: ''  scheme: ''  tcpSocket:  port: ''  initialDelaySeconds: {}  timeoutSeconds: {}  readinessProbe:  exec:  command:  - ''  httpGet:  path: ''  port: ''  host: ''  scheme: ''  tcpSocket:  port: ''  initialDelaySeconds: {}  timeoutSeconds: {}  lifecycle:  postStart:  exec:  command:  - ''  httpGet:  path: ''  port: ''  host: ''  scheme: ''  tcpSocket:  port: ''  preStop:  exec:  command:  - ''  httpGet:  path: ''  port: ''  host: ''  scheme: ''  tcpSocket:  port: ''  terminationMessagePath: ''  imagePullPolicy: ''  securityContext:  capabilities:  add:  - {}  drop:  - {}  privileged: false  seLinuxOptions:  user: ''  role: ''  type: ''  level: ''  runAsUser: {}  runAsNonRoot: false  stdin: false  stdinOnce: false  tty: false  restartPolicy: ''  terminationGracePeriodSeconds: {}  activeDeadlineSeconds: {}  dnsPolicy: ''  nodeSelector: any  host: ''  serviceAccountName: ''  serviceAccount: ''  nodeName: ''  hostNetwork: false  hostPID: false  hostIPC: false  securityContext:  seLinuxOptions:  user: ''  role: ''  type: ''  level: ''  runAsUser: {}  runAsNonRoot: false  supplementalGroups:  - {}  fsGroup: {}  imagePullSecrets:  - name: ''  status:  replicas: {}  observedGeneration: {} |

### 范例3（较全）

|  |
| --- |
| apiVersion: v1  kind: List  items:  - apiVersion: v1    kind: ReplicationController    metadata:      name: pweb      namespace: wangjin      labels:        app: pweb    spec:      replicas: 2      selector:        app: pweb #要与下面template.labels内容对应      template:        metadata:          name: pweb          labels:            app: pweb #容器的标签        spec:          containers:          - image: registry.example.com:5000/wangjin/pweb:wj-pweb-20170612-v2            name: pweb            ports: #多个端口时候，ports只需要写一次            - containerPort: 9080 #容器内部port              name: app-port #多个port时候，要写name区分              protocol: TCP            - containerPort: 9060              name: was-port              protocol: TCP            - containerPort: 8880              name: soap-port              protocol: TCP            lifecycle:               preStop: #容器关闭前，进行的动作，可以自定义脚本                 exec:                   command:                   - sh                   - /home/delete-sql.sh            volumeMounts: #把volume挂载到容器内部            - mountPath: /home/netbank/share #容器内的路径              name: share #使用的volume名称，需要与下面的volume name对应            - mountPath: /home/netbank/facenas              name: facenas            - mountPath: /logbak              name: logbak            - mountPath: /mnt              name: log            readinessProbe: #监控检查              exec:                command: ["curl","http://localhost:9060/ibm/console/unsecureLogon.jsp"]              failureThreshold: 2              initialDelaySeconds: 120              timeoutSeconds: 5          volumes: #pvc挂载到容器          - name: share            persistentVolumeClaim:              claimName: pweb-share #pvc名称          - name: facenas            persistentVolumeClaim:              claimName: pweb-facenas          - name: logbak            persistentVolumeClaim:              claimName: pweb-logbak          - name: log            persistentVolumeClaim:              claimName: pweb-log |

注意项补充

#### ports，volumeMounts，volumes

ports，volumeMounts，volumes 这些每项下面有多条时候，这几个参数只要写一行，如果写成下面这种，将只有最后一个volumes生效。

每项下面的条目要加name参数以区分。

volumes与containers平级，书写时候要注意写到下面。如果把readiness写到volumes下面，那么readiness将不会生效。

|  |
| --- |
| volumes:  - name: share  persistentVolumeClaim:  claimName: pweb-share  volumes:  - name: facenas  persistentVolumeClaim:  claimName: pweb-facenas  volumes:  - name: logbak  persistentVolumeClaim:  claimName: pweb-logbak  volumes:  - name: log  persistentVolumeClaim:  claimName: pweb-log |

#### lifecycle

与image缩进平级

|  |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: lifecycle-demo  spec:  containers:  - name: lifecycle-demo-container  image: nginx  lifecycle:  postStart:  exec:  command: ["/bin/sh", "-c", "echo Hello from the postStart handler > /usr/share/message"]  preStop:  exec:  command: ["/usr/sbin/nginx","-s","quit"] |

可以定义容器启动停止前的动作

#### command

可以替换容器中的CMD

|  |
| --- |
| apiVersion: v1 kind: Pod metadata:   name: pod-w-message spec:   containers:   - name: messager     image: "ubuntu:14.04"     command: ["/bin/sh","-c"]      args: ["sleep 60 && /bin/echo Sleep expired > /dev/termination-log"] |

|  |
| --- |
| apiVersion: v1  kind: Pod  metadata:  labels:  name: test-hostpath  role: master  name: test-hostpath  spec:  containers:  - name: test-hostpath  image: registry:5000/back\_demon:1.0  command:  - /run.sh |

#### Volumes

pod中可以直接定义挂载到nfs，hostpath，emptydir等

|  |
| --- |
| volumes:  - name: rethinkdb-storage  emptyDir: {} |

|  |
| --- |
| volumes:  - name: ssl-certs  hostPath:  path: /etc/ssl/certs |

|  |
| --- |
| volumes:  - name: nfs-storage  nfs:  server: 192.168.20.47  path: "/data/disk1" |

|  |
| --- |
| volumes:  - name: secret  secret:  secretName: mysecret |

#### nodeSelector

可以指定发布到某个node节点。

前提对节点打标签

oc label node node1.example.com app=nginxplus

|  |
| --- |
| apiVersion: v1  kind: ReplicationController  metadata:  name: nginxplus-rc  spec:  replicas: 1  selector:  app: nginxplus  template:  metadata:  labels:  app: nginxplus  spec:  nodeSelector:  role: nginxplus  containers:  - name: nginxplus  imagePullPolicy: IfNotPresent  image: nginxplus  ports:  - name: http  containerPort: 80  hostPort: 80  - name: http-alt  containerPort: 8080  hostPort: 8080 |

#### resources

对容器做资源限制

|  |
| --- |
| apiVersion: v1  kind: ReplicationController  metadata:  name: perbank  labels:  app: perbank  spec:  replicas: 1  selector:  app: perbank  template:  metadata:  name: perbank  labels:  app: perbank  spec:  containers:  - image: registry.example.com:5000/perbank:f01  name: perbank  ports:  - containerPort: 9080  name: app-port  protocol: TCP  - containerPort: 9060  name: was-port  protocol: TCP  resources: #限制container的资源  requests: #请求的资源  cpu: 800m #1000m就是1核  memory: 2Gi  limits: #最大可以使用的资源  cpu: 1600m  memory: 4Gi |

## pod的liveness & readiness

liveness & readiness是pod中的参数，但内容较多，单独拿出来。

Kubelet使用liveness probe（存活探针）来确定何时重启容器。例如，当应用程序处于运行状态但无法做进一步操作，liveness探针将捕获到deadlock，重启处于该状态下的容器，使应用程序在存在bug的情况下依然能够继续运行下去。

Kubelet使用readiness probe（就绪探针）来确定容器是否已经就绪可以接受流量。只有当Pod中的容器都处于就绪状态时kubelet才会认定该Pod处于就绪状态。该信号的作用是控制哪些Pod应该作为service的后端。如果Pod处于非就绪状态，那么它们将会被从service的load balancer中移除。

有三种liveness/readiness probe可供选择：http、tcp和container exec。前两者检查Kubernetes是否能使http或者tcp连接至指定端口。container exec probe可用来运行容器里的指定命令，并维护容器里的停止响应代码。如下所示的代码片段中,我们使用http probe发送Root URL，使用 GET请求到端口80。

liveness和readiness在参数格式上用法相同，只是作用不通。

### exec

|  |
| --- |
| apiVersion: v1  kind: Pod  metadata:  labels:  test: liveness  name: liveness-exec  spec:  containers:  - name: liveness  args:  - /bin/sh  - -c  - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600  image: gcr.io/google\_containers/busybox  livenessProbe:  exec:  command:  - cat  - /tmp/healthy  initialDelaySeconds: 5  periodSeconds: 5 |

periodSeconds 规定kubelet要每隔5秒执行一次liveness probe。

initialDelaySeconds 告诉kubelet在第一次执行probe之前要的等待5秒钟。探针检测命令是在容器中执行 cat /tmp/healthy 命令。如果命令执行成功，将返回0，kubelet就会认为该容器是活着的并且很健康。如果返回非0值，kubelet就会杀掉这个容器并重启它。

### httpget

|  |
| --- |
| apiVersion: v1  kind: Pod  metadata:  labels:  test: liveness  name: liveness-http  spec:  containers:  - name: liveness  args:  - /server  image: gcr.io/google\_containers/liveness  livenessProbe:  httpGet:  path: /healthz  port: 8080  httpHeaders:  - name: X-Custom-Header  value: Awesome  initialDelaySeconds: 3  periodSeconds: 3 |

livenessProbe 指定kubelete需要每隔3秒执行一次liveness probe。initialDelaySeconds 指定kubelet在该执行第一次探测之前需要等待3秒钟。该探针将向容器中的server的8080端口发送一个HTTP GET请求。如果server的/healthz路径的handler返回一个成功的返回码，kubelet就会认定该容器是活着的并且很健康。如果返回失败的返回码，kubelet将杀掉该容器并重启它。

任何大于200小于400的返回码都会认定是成功的返回码。其他返回码都会被认为是失败的返回码。

### 更多参数配置

Probe中有很多精确和详细的配置，通过它们你能准确的控制liveness和readiness检查：

initialDelaySeconds：容器启动后第一次执行探测是需要等待多少秒。

periodSeconds：执行探测的频率。默认是10秒，最小1秒。

timeoutSeconds：探测超时时间。默认1秒，最小1秒。

successThreshold：探测失败后，最少连续探测成功多少次才被认定为成功。默认是1。对于liveness必须是1。最小值是1。

failureThreshold：探测成功后，最少连续探测失败多少次才被认定为失败。默认是3。最小值是1。

HTTP probe中可以给 httpGet设置其他配置项：

host：连接的主机名，默认连接到pod的IP。你可能想在http header中设置"Host"而不是使用IP。

scheme：连接使用的schema，默认HTTP。

path: 访问的HTTP server的path。

httpHeaders：自定义请求的header。HTTP运行重复的header。

port：访问的容器的端口名字或者端口号。端口号必须介于1和65525之间。

对于HTTP探测器，kubelet向指定的路径和端口发送HTTP请求以执行检查。 Kubelet将probe发送到容器的IP地址，除非地址被httpGet中的可选host字段覆盖。 在大多数情况下，你不想设置主机字段。 有一种情况下你可以设置它。 假设容器在127.0.0.1上侦听，并且Pod的hostNetwork字段为true。 然后，在httpGet下的host应该设置为127.0.0.1。 如果你的pod依赖于虚拟主机，这可能是更常见的情况，你不应该是用host，而是应该在httpHeaders中设置Host头。

## ReplicaSet（rs）

在新版本的Kubernetes中建议使用ReplicaSet来取代ReplicationCtronller。ReplicaSet跟ReplicationCtronller没有本质的不同，只是名字不一样，并且ReplicaSet支持集合式的selector。

虽然ReplicaSet可以独立使用，但一般还是建议使用 Deployment 来自动管理ReplicaSet，这样就无需担心跟其他机制的不兼容问题（比如ReplicaSet不支持rolling-update但Deployment支持）。

pod部分的书写规则与rc相同，更多细节可参考rc部分。

|  |
| --- |
| apiVersion: extensions/v1beta1  kind: ReplicaSet  metadata:  name: frontend  # these labels can be applied automatically  # from the labels in the pod template if not set  # labels:  # app: guestbook  # tier: frontend  spec:  # this replicas value is default  # modify it according to your case  replicas: 3  # selector can be applied automatically  # from the labels in the pod template if not set,  # but we are specifying the selector here to  # demonstrate its usage.  selector:  matchLabels:  tier: frontend  matchExpressions:  - {key: tier, operator: In, values: [frontend]}  template:  metadata:  labels:  app: guestbook  tier: frontend  spec:  containers:  - name: php-redis  image: gcr.io/google\_samples/gb-frontend:v3  resources:  requests:  cpu: 100m  memory: 100Mi  env:  - name: GET\_HOSTS\_FROM  value: dns  # If your cluster config does not include a dns service, then to  # instead access environment variables to find service host  # info, comment out the 'value: dns' line above, and uncomment the  # line below.  # value: env  ports:  - containerPort: 80 |

## [Deployment](http://blog.csdn.net/liukuan73/article/details/54710766)（deploy）

Deployment为Pod和ReplicaSet提供了一个声明式定义(declarative)方法，用来替代以前的ReplicationController来方便的管理应用。典型的应用场景包括：

定义Deployment来创建Pod和ReplicaSet

滚动升级和回滚应用

扩容和缩容

暂停和继续Deployment

pod部分的书写规则与rc相同，更多细节可参考rc部分。

|  |
| --- |
| apiVersion: extensions/v1beta1  kind: Deployment  metadata:  name: nginx-deployment  spec:  replicas: 3  template:  metadata:  labels:  app: nginx  spec:  containers:  - name: nginx  image: nginx:1.7.9  ports:  - containerPort: 80 |

## Pod

单独创建的pod多用于测试验证用途

### ubuntu

镜像也可以是centos，rhel。注意系统镜像需要一个command来保持容器运行

|  |
| --- |
| apiVersion: v1  kind: Pod  metadata:    name: hello-world  spec:  # specification of the pod’s contents    restartPolicy: Never    containers:    - name: hello      image: "ubuntu:14.04"      env: # 这是一个环境变量      - name: MESSAGE        value: "hello world"      command: ["/bin/sh","-c"] # Kubernetes并不会自动运行shell，需要手动指定      args: ["/bin/echo \"${MESSAGE}\""] # 这是用到的环境变量展开 |

### nginx

tomcat同理。

注意hostport，这个pod将会占用对应的node节点上的9088端口

|  |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: nginx-testing  spec:  containers:  - name: nginx-container  image: docker.io/nginx  ports:  - containerPort: 80  hostPort: 9088 |

### pod中多容器

|  |
| --- |
| apiVersion: v1  kind: Pod  metadata:   name: rss-site   labels:     app: web  spec:   containers:     - name: front-end       image: nginx       ports:         - containerPort: 80     - name: rss-reader       image: nickchase/rss-php-nginx:v1       ports:         - containerPort: 88 |

## job

Job负责批处理任务，即仅执行一次的任务，它保证批处理任务的一个或多个Pod成功结束。

Job Spec格式

spec.template格式同Pod

RestartPolicy仅支持Never或OnFailure

单个Pod时，默认Pod成功运行后Job即结束

.spec.completions标志Job结束需要成功运行的Pod个数，默认为1

.spec.parallelism标志并行运行的Pod的个数，默认为1

spec.activeDeadlineSeconds标志失败Pod的重试最大时间，超过这个时间不会继续重试

一定记得给Job配置activeDeadlineSeconds，防止问题job无限制的create pod。参数查找rc范例。

一个简单的例子：

|  |
| --- |
| apiVersion: batch/v1  kind: Job  metadata:  name: pi  spec:  template:  metadata:  name: pi  spec:  containers:  - name: pi  image: perl  command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]  restartPolicy: Never  $ kubectl create -f ./job.yaml  job "pi" created  $ pods=$(kubectl get pods --selector=job-name=pi --output=jsonpath={.items..metadata.name})  $ kubectl logs $pods  3.141592653589793238462643383279502... |

固定结束次数的Job示例：

|  |
| --- |
| apiVersion: batch/v1  kind: Job  metadata:  name: busybox  spec:  completions: 3  template:  metadata:  name: busybox  spec:  containers:  - name: busybox  image: busybox  command: ["echo", "hello"]  restartPolicy: Never |

## template

template是openshift特有的模版资源。

创建template的时候，文件中的资源不会生成，只有通过template发布的时候，定义资源才会生效。

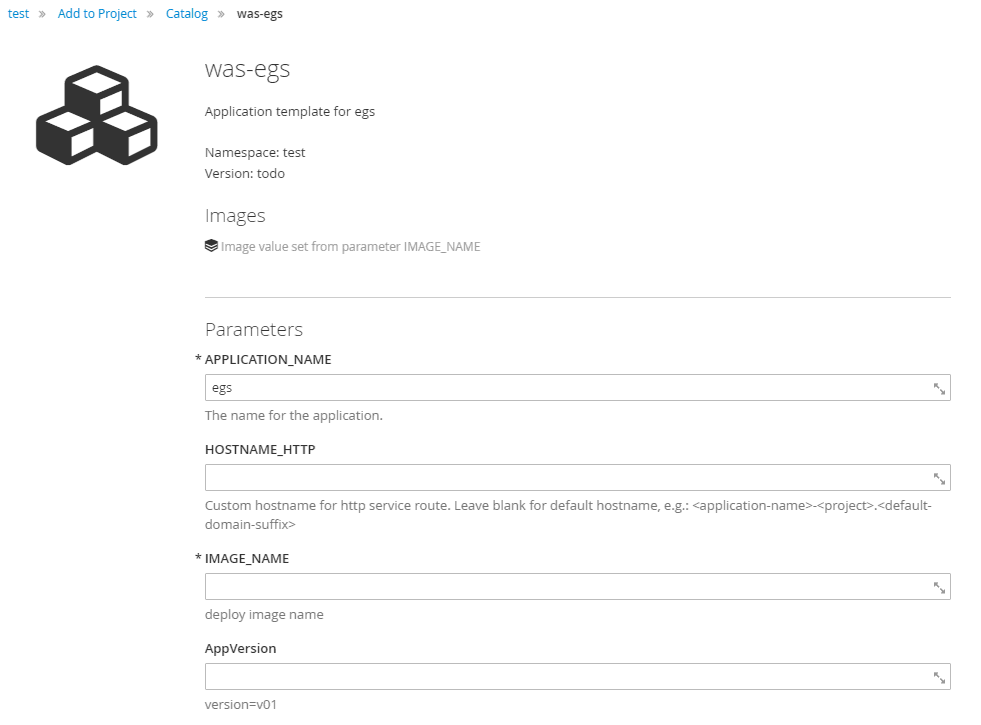
template中可以定义多个资源。资源中可以使用变量，通过parameters，在发布的时候传递到资源里。

DeploymentConfig下pod资源的定义同rc。

parameters中定义的内容，在通过模版发布的时候会展现出来。

|  |
| --- |
| apiVersion: v1  kind: Template  labels:  template: egs #标签，可以多条  vender: jsbchina  metadata:  annotations:  description: Application template for egs  iconClass: icon-websphere #图标  tags: was,java,jboss,xpass  version: todo  creationTimestamp: null  name: was-egs #template名称  objects:  - apiVersion: v1  kind: Service  metadata:  annotations:  description: The web server's http port.  labels:  application: ${APPLICATION\_NAME} #变量见parameters，发布的时候传递进来  name: ${APPLICATION\_NAME}  spec:  ports:  - name: was-port  nodePort: 30960  port: 9060  protocol: TCP  targetPort: was-port  - name: egs-port  nodePort: 30980  port: 9080  protocol: TCP  targetPort: egs-port  selector:  application: ${APPLICATION\_NAME}  type: NodePort  - apiVersion: v1  id: ${APPLICATION\_NAME}-http  kind: Route  metadata:  annotations:  description: Route for application's http service.  labels:  application: ${APPLICATION\_NAME}  name: ${APPLICATION\_NAME}  spec:  host: ${HOSTNAME\_HTTP}  port:  targetPort: was-port  to:  kind: Service  name: ${APPLICATION\_NAME}  - apiVersion: v1  kind: DeploymentConfig  metadata:  labels:  application: ${APPLICATION\_NAME}  name: ${APPLICATION\_NAME}  spec:  replicas: ${replica} #pod数量，发布时导入  selector:  deploymentConfig: ${APPLICATION\_NAME}  strategy:  type: Recreate  template:  metadata:  labels:  application: ${APPLICATION\_NAME}  deploymentConfig: ${APPLICATION\_NAME}  name: ${APPLICATION\_NAME}  spec:  containers:  - env:  - name: AppName\_version  value: ${AppVersion} #环境变量的值发布时导入  image: ${IMAGE\_NAME} #镜像名称发布时导入  imagePullPolicy: Always  name: ${APPLICATION\_NAME}  ports:  - containerPort: 9080  name: egs-port  protocol: TCP  - containerPort: 9060  name: was-port  protocol: TCP  triggers:  - type: ConfigChange  parameters: #以下内容将显示在应用发布时页面上  - description: The name for the application. #每个description开始，为一项  name: APPLICATION\_NAME  required: true #true时为必填项  value: egs #默认值  - description: 'Custom hostname for http service route. Leave blank for default hostname,  e.g.: <application-name>-<project>.<default-domain-suffix>'  name: HOSTNAME\_HTTP  - description: deploy image name  name: IMAGE\_NAME  required: true  - description: version=v01  name: AppVersion  - description: The number of Pods to create replcaia=1  name: replica |

应用发布页面，parameters定义的内容都会在此显示



## Service(svc)

Service是对一组提供相同功能的Pods的抽象，并为它们提供一个统一的入口。借助Service，应用可以方便的实现服务发现与负载均衡，并实现应用的零宕机升级。Service通过标签来选取服务后端，一般配合Replication Controller或者Deployment来保证后端容器的正常运行。

Service有三种类型：

ClusterIP：默认类型，自动分配一个仅cluster内部可以访问的虚拟IP

NodePort：在ClusterIP基础上为Service在每台机器上绑定一个端口，这样就可以通过<NodeIP>:NodePort来访问改服务

LoadBalancer：在NodePort的基础上，借助cloud provider创建一个外部的负载均衡器，并将请求转发到<NodeIP>:NodePort

### ClusterIP

|  |
| --- |
| apiVersion: v1  kind: Service  metadata:    name: nginxsvc    labels:      app: nginx  spec:    ports:    - port: 80 #访问的地址为 http://clusterIP:80/      protocol: TCP    selector: #意思是由带“app=nginx” label的pod来处理      app: nginx |

### nodeport

访问方式： 任意节点IP+nodeport

|  |
| --- |
| - apiVersion: v1  kind: Service  metadata:  annotations:  description: The web server's http port.  namespace: wangjin  labels:  app: pweb  name: pweb  spec:  ports:  - name: was-port  nodePort: 31001  port: 9060  protocol: TCP  targetPort: 9060  - name: app-port  nodePort: 31002  port: 9080  protocol: TCP  targetPort: 9080  - name: soap-port  nodePort: 31003  port: 8880  protocol: TCP  targetPort: 8880  selector:  app: pweb  sessionAffinity: ClientIP #相当于IP hash  type: NodePort |

### loadBalancer

|  |
| --- |
| kind: Service  apiVersion: v1  metadata:  name: my-service  spec:  selector:  app: MyApp  ports:  - protocol: TCP  port: 80  targetPort: 9376  nodePort: 30061  clusterIP: 10.0.171.239  loadBalancerIP: 78.11.24.19  type: LoadBalancer  status:  loadBalancer:  ingress:  - ip: 146.148.47.155 |

## Persistentvolumes(pv)

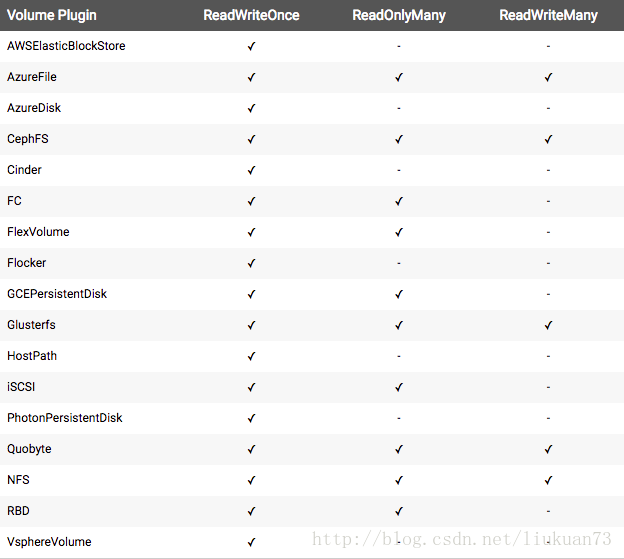
PersistentVolume（PV）是集群之中的一块网络存储。跟 Node 一样，也是集群的资源。PV 跟 Volume (卷) 类似，不过会有独立于 Pod 的生命周期。pv是全局的，不区分namespace。

### 访问模式

第一种，ReadWriteOnce：是最基本的方式，可读可写，但只支持被单个Pod挂载。

第二种，ReadOnlyMany：可以以只读的方式被多个Pod挂载。

第三种，ReadWriteMany：这种存储可以以读写的方式被多个Pod共享。不是每一种存储都支持这三种方式，像共享方式，目前支持的还比较少，比较常用的是NFS。在PVC绑定PV时通常根据两个条件来绑定，一个是存储的大小，另一个就是访问模式。



### 回收策略

现有回收策略有：

Retain – 手动重新使用

Recycle – 基本的删除操作 (“rm -rf /thevolume/\*”)

Delete – 关联的后端存储卷一起删除，后端存储例如AWS EBS, GCE PD或OpenStack Cinder

目前只有NFS和HostPath支持回收，AWS EBS, GCE PD和Cinder volumes只支持删除。

### 卷的状态

卷有四种状态，一个卷必属于其中之一:

Available –闲置状态，没有被绑定到PVC

Bound – 绑定到PVC

Released – PVC被删掉，资源没有被在利用

Failed – 自动回收失败

### nfs 范例

|  |
| --- |
| apiVersion: v1  kind: PersistentVolume  metadata:  name: pweb-nas  labels:  name: pweb-nas  spec:  accessModes:  - ReadWriteMany  capacity:  storage: 100Gi  nfs:  path: /vx/nfs-share  server: 77.1.1.2  persistentVolumeReclaimPolicy: Retain |

NFS server需要注意权限设定。

|  |
| --- |
| export volname=cassandra  mkdir -p /srv/nfs/${volname}  chown nfsnobody:nfsnobody /srv/nfs/${volname}  chmod 700 /srv/nfs/${volname}  echo "/srv/nfs/${volname} \*(rw,sync,all\_squash)">> /etc/exports  systemctl restart rpcbind nfs-server nfs-lock nfs-idmap  systemctl enable nfs-server |

### hostpath范例

|  |
| --- |
| kind: PersistentVolume  apiVersion: v1  metadata:  name: pv0001  labels:  type: local  spec:  capacity:  storage: 10Gi  accessModes:  - ReadWriteOnce  hostPath:  path: "/tmp/data01" |

hostpath后续pod调用时候，会直接挂载到node节点的本地目录，比如/tmp/data01。需要修改下目录的selinux 上下文。不然会没权限。

|  |
| --- |
| chcon -Rt svirt\_sandbox\_file\_t /tmp/data01 |

## Persistentvolumeclaims(pvc)

PersistentVolumeClaim（一些简称PVC）：是Namespace里的资源，描述对PV的一个请求。请求信息包含存储大小，访问模式等。

pvc创建后会自动与Available状态的pv关联。

pvc的容量要小于等于pv的，才能自动关联。如果PV 20G，PVC 请求容量10G。关联后最终PVC为20G。

也可以通过selector 指定PV，见范例2

### 范例1

|  |
| --- |
| apiVersion: v1  kind: PersistentVolumeClaim  metadata:  name: myclaim  namespace: test  spec:  accessModes:  - ReadWriteOnce  resources:  requests:  storage: 5Gi |

### 范例2

|  |
| --- |
| apiVersion: v1  kind: PersistentVolumeClaim  metadata:    name: pweb-nas    namespace: wangjin    labels:      name: pweb-nas  spec:    accessModes:      - ReadWriteMany    resources:      requests:        storage: 100Gi    selector:      matchLabels:        name: pweb-nas |

## 对接glusterfs

glusterfs部署方案见文档：Gluster安装及与Openshift集成

官方文档：

https://docs.openshift.com/container-platform/3.4/install\_config/persistent\_storage/persistent\_storage\_glusterfs.html

|  |
| --- |
| kind: Endpoints  apiVersion: v1  metadata:  name: glusterfs-cluster  namespace: xxxxxxxxx  subsets:  - addresses:  - ip: 10.20.112.201  ports:  - port: 1  protocol: TCP  - addresses:  - ip: 10.20.112.202  ports:  - port: 1  protocol: TCP  - addresses:  - ip: 10.20.112.203  ports:  - port: 1  protocol: TCP  - addresses:  - ip: 10.20.112.204  ports:  - port: 1  protocol: TCP  apiVersion: v1  kind: Service  metadata:  name: glusterfs-cluster  spec:  ports:  - port: 1  service和endpoint名字要相同  apiVersion: v1  kind: PersistentVolume  metadata:  name: gluster-map-pv  labels:  name: gluster-map-pv  spec:  capacity:  storage: 1500Gi  accessModes:  - ReadWriteMany  glusterfs:  endpoints: glusterfs-cluster  path: /volume-data  readOnly: false  persistentVolumeReclaimPolicy: Retain  ---  apiVersion: v1  kind: PersistentVolumeClaim  metadata:  name: gluster-map-claim  namespace: shikongxinxi  spec:  accessModes:  - ReadWriteMany  resources:  requests:  storage: 1500Gi  selector:  matchLabels:  name: gluster-map-pv  apiVersion: v1  kind: PersistentVolume  metadata:  name: gluster-redis-pv  labels:  name: gluster-redis-pv  spec:  capacity:  storage: 200Gi  accessModes:  - ReadWriteMany  glusterfs:  endpoints: gluster-endpoints  path: /volume-data  readOnly: false  persistentVolumeReclaimPolicy: Retain  ---  apiVersion: v1  kind: PersistentVolumeClaim  metadata:  name: gluster-redis-claim  namespace: shikongxinxi  spec:  accessModes:  - ReadWriteMany  resources:  requests:  storage: 200Gi  selector:  matchLabels:  name: gluster-redis-pv  apiVersion: v1  kind: Pod  metadata:  name: tomtestgfs  labels:  name: tomcat-1  spec:  containers:  - image: registry.example.com:5000/tomcat7-skxx:v2.3  imagePullPolicy: Always  name: tomcat-skxx235  ports:  - containerPort: 8080  name: http  volumeMounts:  - mountPath: /tmp  name: gfs  securityContext:  privileged: true  supplementalGroups: [500]  volumes:  - name: gfs  persistentVolumeClaim:  claimName: gluster-map-claim |

## Horizontal Pod Autoscaling（hpa）

应用的资源使用率通常都有高峰和低谷的时候，如何削峰填谷，提高集群的整体资源利用率，让service中的Pod个数自动调整呢？这就有赖于Horizontal Pod Autoscaling了，顾名思义，使Pod水平自动缩放。

Horizontal Pod Autoscaling仅适用于Deployment和ReplicationController（ReplicaSet已经被ReplicationController取代），在V1版本中仅支持根据Pod的CPU利用率扩所容，在v1alpha版本中，支持根据内存和用户自定义的metric扩缩容。

通过命令创建

|  |
| --- |
| kubectl autoscale deployment foo --min=2 --max=5 --cpu-percent=80 |

通过文件创建

|  |
| --- |
| apiVersion: autoscaling/v1  kind: HorizontalPodAutoscaler  metadata:  name: ebs  labels:  app: ebs  spec:  maxReplicas: 8  minReplicas: 1  scaleTargetRef:  apiVersion: v1  kind: DeploymentConfig #弹性的对象，也可以是rc  name: ebs #对应资源的名称  targetCPUUtilizationPercentage: 80 |

## configmap（cm）

ConfigMap API资源用来保存key-value pair配置数据，这个数据可以在pods里使用，或者被用来为像controller一样的系统组件存储配置数据。虽然ConfigMap跟[Secrets](https://kubernetes.io/docs/user-guide/secrets/)类似，但是ConfigMap更方便的处理不含敏感信息的字符串。 注意：ConfigMaps不是属性配置文件的替代品。ConfigMaps只是作为多个properties文件的引用。你可以把它理解为Linux系统中的/etc目录，专门用来存储配置文件的目录。下面举个例子，使用ConfigMap配置来创建Kuberntes Volumes，ConfigMap中的每个data项都会成为一个新文件。

|  |
| --- |
| kind: ConfigMap  apiVersion: v1  metadata:  creationTimestamp: 2016-02-18T19:14:38Z  name: example-config  namespace: default  data:  example.property.1: hello  example.property.2: world  example.property.file: |-  property.1=value-1  property.2=value-2  property.3=value-3 |

data一栏包括了配置数据，ConfigMap可以被用来保存单个属性，也可以用来保存一个配置文件。 配置数据可以通过很多种方式在Pods里被使用。ConfigMaps可以被用来：

设置环境变量的值

在容器里设置命令行参数

在数据卷里面创建config文件

用户和系统组件两者都可以在ConfigMap里面存储配置数据。

其实不用看下面的文章，直接从kubectl create configmap -h的帮助信息中就可以对ConfigMap究竟如何创建略知一二了。

|  |
| --- |
| Examples:  # Create a new configmap named my-config based on folder bar  kubectl create configmap my-config --from-file=path/to/bar  # Create a new configmap named my-config with specified keys instead of file basenames on disk  kubectl create configmap my-config --from-file=key1=/path/to/bar/file1.txt --from-file=key2=/path/to/bar/file2.txt  # Create a new configmap named my-config with key1=config1 and key2=config2  kubectl create configmap my-config --from-literal=key1=config1 --from-literal=key2=config2 |

### 创建ConfigMaps

可以使用该命令，用给定值、文件或目录来创建ConfigMap。

|  |
| --- |
| # kubectl create configmap |

#### 使用目录创建

比如我们已经有个了包含一些配置文件，其中包含了我们想要设置的ConfigMap的值：

|  |
| --- |
| $ ls docs/user-guide/configmap/kubectl/  game.properties  ui.properties  $ cat docs/user-guide/configmap/kubectl/game.properties  enemies=aliens  lives=3  enemies.cheat=true  enemies.cheat.level=noGoodRotten  secret.code.passphrase=UUDDLRLRBABAS  secret.code.allowed=true  secret.code.lives=30  $ cat docs/user-guide/configmap/kubectl/ui.properties  color.good=purple  color.bad=yellow  allow.textmode=true  how.nice.to.look=fairlyNice |

使用下面的命令可以创建一个包含目录中所有文件的ConfigMap。

|  |
| --- |
| $ kubectl create configmap game-config --from-file=docs/user-guide/configmap/kubectl |

—from-file指定在目录下的所有文件都会被用在ConfigMap里面创建一个键值对，键的名字就是文件名，值就是文件的内容。

让我们来看一下这个命令创建的ConfigMap：

|  |
| --- |
| $ kubectl describe configmaps game-config  Name: game-config  Namespace: default  Labels: <none>  Annotations: <none>  Data  ====  game.properties: 158 bytes  ui.properties: 83 bytes |

我们可以看到那两个key是从kubectl指定的目录中的文件名。这些key的内容可能会很大，所以在kubectl describe的输出中，只能够看到键的名字和他们的大小。 如果想要看到键的值的话，可以使用kubectl get：

|  |
| --- |
| $ kubectl get configmaps game-config -o yaml |

我们以yaml格式输出配置。

|  |
| --- |
| apiVersion: v1  data:  game.properties: |  enemies=aliens  lives=3  enemies.cheat=true  enemies.cheat.level=noGoodRotten  secret.code.passphrase=UUDDLRLRBABAS  secret.code.allowed=true  secret.code.lives=30  ui.properties: |  color.good=purple  color.bad=yellow  allow.textmode=true  how.nice.to.look=fairlyNice  kind: ConfigMap  metadata:  creationTimestamp: 2016-02-18T18:34:05Z  name: game-config  namespace: default  resourceVersion: "407"  selfLink: /api/v1/namespaces/default/configmaps/game-config  uid: 30944725-d66e-11e5-8cd0-68f728db1985 |

#### 使用文件创建

刚才使用目录创建的时候我们—from-file指定的是一个目录，只要指定为一个文件就可以从单个文件中创建ConfigMap。

|  |
| --- |
| $ kubectl create configmap game-config-2 --from-file=docs/user-guide/configmap/kubectl/game.properties  $ kubectl get configmaps game-config-2 -o yaml  apiVersion: v1  data:  game-special-key: |  enemies=aliens  lives=3  enemies.cheat=true  enemies.cheat.level=noGoodRotten  secret.code.passphrase=UUDDLRLRBABAS  secret.code.allowed=true  secret.code.lives=30  kind: ConfigMap  metadata:  creationTimestamp: 2016-02-18T18:54:22Z  name: game-config-3  namespace: default  resourceVersion: "530"  selfLink: /api/v1/namespaces/default/configmaps/game-config-3  uid: 05f8da22-d671-11e5-8cd0-68f728db1985 |

—from-file这个参数可以使用多次，你可以使用两次分别指定上个实例中的那两个配置文件，效果就跟指定整个目录是一样的。

#### 使用Literal值创建

使用文字值创建，利用—from-literal参数传递配置信息，该参数可以使用多次，格式如下；

|  |
| --- |
| $ kubectl create configmap special-config --from-literal=special.how=very --from-literal=special.type=charm  $ kubectl get configmaps special-config -o yaml  apiVersion: v1  data:  special.how: very  special.type: charm  kind: ConfigMap  metadata:  creationTimestamp: 2016-02-18T19:14:38Z  name: special-config  namespace: default  resourceVersion: "651"  selfLink: /api/v1/namespaces/default/configmaps/special-config  uid: dadce046-d673-11e5-8cd0-68f728db1985 |

### Pod中使用ConfigMap

#### 使用ConfigMap来替代环境变量

ConfigMap可以被用来填入环境变量。看下下面的ConfigMap。

|  |
| --- |
| apiVersion: v1  kind: ConfigMap  metadata:  name: special-config  namespace: default  data:  special.how: very  special.type: charm  apiVersion: v1  kind: ConfigMap  metadata:  name: env-config  namespace: default  data:  log\_level: INFO |

我们可以在Pod中这样使用ConfigMap：

|  |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: dapi-test-pod  spec:  containers:  - name: test-container  image: gcr.io/google\_containers/busybox  command: [ "/bin/sh", "-c", "env" ]  env:  - name: SPECIAL\_LEVEL\_KEY  valueFrom:  configMapKeyRef:  name: special-config  key: special.how  - name: SPECIAL\_TYPE\_KEY  valueFrom:  configMapKeyRef:  name: special-config  key: special.type  envFrom:  - configMapRef:  name: env-config  restartPolicy: Never |

这个Pod运行后会输出如下几行：

SPECIAL\_LEVEL\_KEY=very

SPECIAL\_TYPE\_KEY=charm

log\_level=INFO

#### 用ConfigMap设置命令行参数

ConfigMap也可以被使用来设置容器中的命令或者参数值。它使用的是Kubernetes的$(VAR\_NAME)替换语法。我们看下下面这个ConfigMap。

|  |
| --- |
| apiVersion: v1  kind: ConfigMap  metadata:  name: special-config  namespace: default  data:  special.how: very  special.type: charm |

为了将ConfigMap中的值注入到命令行的参数里面，我们还要像前面那个例子一样使用环境变量替换语法${VAR\_NAME)。（其实这个东西就是给Docker容器设置环境变量，以前我创建镜像的时候经常这么玩，通过docker run的时候指定-e参数修改镜像里的环境变量，然后docker的CMD命令再利用该$(VAR\_NAME)通过sed来来修改配置文件或者作为命令行启动参数。）

|  |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: dapi-test-pod  spec:  containers:  - name: test-container  image: gcr.io/google\_containers/busybox  command: [ "/bin/sh", "-c", "echo $(SPECIAL\_LEVEL\_KEY) $(SPECIAL\_TYPE\_KEY)" ]  env:  - name: SPECIAL\_LEVEL\_KEY  valueFrom:  configMapKeyRef:  name: special-config  key: special.how  - name: SPECIAL\_TYPE\_KEY  valueFrom:  configMapKeyRef:  name: special-config  key: special.type  restartPolicy: Never |

运行这个Pod后会输出：

very charm

#### 通过数据卷插件使用ConfigMap

ConfigMap也可以在数据卷里面被使用。还是这个ConfigMap。

|  |
| --- |
| apiVersion: v1  kind: ConfigMap  metadata:  name: special-config  namespace: default  data:  special.how: very  special.type: charm |

在数据卷里面使用这个ConfigMap，有不同的选项。最基本的就是将文件填入数据卷，在这个文件中，键就是文件名，键值就是文件内容：

|  |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: dapi-test-pod  spec:  containers:  - name: test-container  image: gcr.io/google\_containers/busybox  command: [ "/bin/sh", "-c", "cat /etc/config/special.how" ]  volumeMounts:  - name: config-volume  mountPath: /etc/config  volumes:  - name: config-volume  configMap:  name: special-config  restartPolicy: Never |

运行这个Pod的输出是very。

我们也可以在ConfigMap值被映射的数据卷里控制路径。

|  |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: dapi-test-pod  spec:  containers:  - name: test-container  image: gcr.io/google\_containers/busybox  command: [ "/bin/sh","-c","cat /etc/config/path/to/special-key" ]  volumeMounts:  - name: config-volume  mountPath: /etc/config  volumes:  - name: config-volume  configMap:  name: special-config  items:  - key: special.how  path: path/to/special-key  restartPolicy: Never |

运行这个Pod后的结果是very。

## 多资源共存的yaml文件书写规则

同一个文件中要写rc，service等。kind为list，rc和service每行前面加两个空格。

每行前面加两空格的方法：vi, shift+: , 输入1, s/^/ / 回车

|  |
| --- |
| apiVersion: v1  kind: List  items:  - apiVersion: v1  kind: ReplicationController  metadata:  name: tomcat3  namespace: default  labels:  app: tomcat3  spec:  replicas: 2  selector:  app: tomcat3  template:  metadata:  name: tomcat3  labels:  app: tomcat3  spec:  containers:  - image: tomcat  name: tomcat3  ports:  - containerPort: 8080  name: app-port  protocol: TCP  - apiVersion: v1  kind: Service  metadata:  annotations:  description: The web server's http port.  namespace: default  labels:  app: tomcat3  name: tomcat3  spec:  ports:  - name: was-port  nodePort: 31712  port: 8080  protocol: TCP  targetPort: 8080  selector:  app: tomcat3  type: NodePort |

## 参考文档

https://rootsongjc.gitbooks.io/kubernetes-handbook/content/concepts/

https://docs.openshift.com/container-platform/3.4/welcome/index.html