一、K8S概念

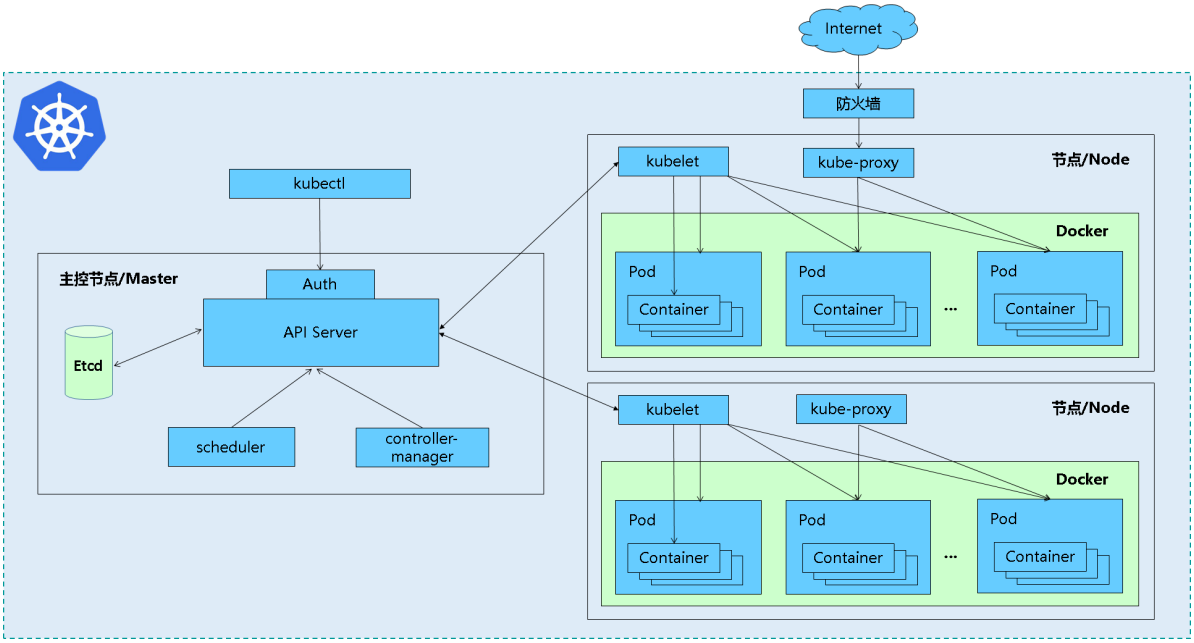
1.说明

- Kubernetes是Google在2014年开源的一个容器集群管理系统，简称K8S。
- Kubernetes主要用于容器化应用部署，扩展和管理，目的是让部署容器应用简单高效。

官网地址：https://kubernetes.io/

官方文档：https://kubernetes.io/zh/docs

2.K8s架构及组件



Master角色

- **kube-apiserver**

  Kubernetes API 集群的统一入口，各组件协调者。所有对象资源的增删改查和监听操作都交给APIServer处理后再提交给Etcd存储。

- **kube-controller-manager**

  处理集群中常规后台任务，一个资源对应一个控制器，而ControllerManager就是负责管理这些控制器的。

- **kube-scheduler**

  根据调度算法为新创建的Pod选择一个Node节点，可以任意部署,可以部署在同一个节点上,也可以部署在不同的节点上。

- **etcd**

  分布式键值存储系统。用于保存集群状态数据，如POD,Service等信息。

Worker Node角色

- **kubelet**

  kubelet是Master在Node节点上的Agent，管理本机运行容器的生命周期，比如创建容器、Pod挂载数据卷、下载secret、获取容器和节点状态等工作。kubelet将每个Pod转换成一组容器。

- **kube-proxy**

  在Node节点上实现Pod网络代理，维护网络规则和四层负载均衡工作。

- **docker**

  容器引擎

3.k8s基本概念

POD

- 最小部署单元
- 一组容器的集合
- 一个Pod中的容器共享网络命名空间
- Pod是短暂的（重新拉取pod ip会发生变化）

Controllers（控制器）

- Deployment： 无状态应用部署
- StatefulSet： 有状态应用部署
- DaemonSet： 确保所有Node运行同一个Pod
- Job： 一次性任务
- Cronjob： 定时任务

控制器是更高级层次对象，用于部署和管理Pod。

Service

- 防止Pod失联
- 定义一组Pod的访问策略

Label： 标签，附加到某个资源上，用于关联对象、查询和筛选

Namespaces： 命名空间，将对象逻辑上隔离

二、部署K8S

参考《二进制部署K8S》文档进行部署

注意部署时关于CNI网络说明：

1. 部署时需要注意根据需求选择CNI网络组件
2. 常用的网络组件
   - flannel（vxlan:隧道方案、hostgw：路由方案，性能最好）
   - calico（ipip、bgp）
3. 如何选择网络组件
   - 查看网络是否有限制。如果有限制建议选择vxlan
   - 根据集群规模大小选择。如果规模小，并且没有网络选择建议选择hostgw，如果大规模集群建议选择calico
   - 有多租户acl（访问限制）需求，选择calico

- 维护成本。flannel相对简单

三、kubectl命令行管理工具

1.kubectl命令说明

kubectl是Kubernetes集群的命令行工具，通过kubectl能够对集群本身进行管理，并能够在集群上进行容器化应用的安装部署。

```
kubectl --help # 查看帮助信息

kubectl create --help # 查看create命令帮助信息
```

命令	描述
create	通过文件名或标准输入创建资源
expose	将一个资源公开为一个新的Service
run	在集群中运行一个特定的镜像
set	在对象上设置特定的功能
get	显示一个或多个资源
explain	文档参考资料
edit	使用默认的编辑器编辑一个资源。
delete	通过文件名、标准输入、资源名称或标签选择器来删除资源。
rollout	管理资源的发布
rolling-update	对给定的复制控制器滚动更新
scale	扩容或缩容Pod数量，Deployment、ReplicaSet、RC或Job
autoscale	创建一个自动选择扩容或缩容并设置Pod数量
certificate	修改证书资源
cluster-info	显示集群信息
top	显示资源（CPU/Memory/Storage）使用。需要Heapster运行
cordon	标记节点不可调度
uncordon	标记节点可调度
drain	驱逐节点上的应用，准备下线维护
taint	修改节点taint标记
describe	显示特定资源或资源组的详细信息
logs	在一个Pod中打印一个容器日志。如果Pod只有一个容器，容器名称是可选的
attach	附加到一个运行的容器
exec	执行命令到容器
port-forward	转发一个或多个本地端口到一个pod
proxy	运行一个proxy到Kubernetes API server
cp	拷贝文件或目录到容器中
auth	检查授权
apply	通过文件名或标准输入对资源应用配置
patch	使用补丁修改、更新资源的字段
replace	通过文件名或标准输入替换一个资源

命令	描述
convert	不同的API版本之间转换配置文件
label	更新资源上的标签
annotate	更新资源上的注释
completion	用于实现kubectl工具自动补全
api-versions	打印受支持的API版本
config	修改kubeconfig文件（用于访问API，比如配置认证信息）
help	所有命令帮助
plugin	运行一个命令行插件
version	打印客户端和服务版本信息

2 使用kubectl管理应用生命周示例

- 创建pod

```
# 创建一个deplomentt，web为名称，--image指定镜像
kubectl create deployment web --image=nginx
# 查看
kubectl get deploy,pods
```

```
[root@k8s-master1 ~]# kubectl get deploy,pods
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/web   1/1     1            1           6h33m

NAME                       READY   STATUS    RESTARTS   AGE
pod/web-5dcb957ccc-tsh7g   1/1     Running   0          6h33m
[root@k8s-master1 ~]#
```

- 发布

```
# 把创建的pod暴露出去
kubectl  expose deployment web --port=18080 --type=NodePort --target-port=80 --
name=web
# 参数说明：
# deployment web  创建的deployment 名称
# --port=18080 : service端口
# --type=NodePort : 访问方式,这里是NodePort
# --target-port=80 : 容器端口

# 查看service
kubectl  get service
```

```
[root@k8s-master1 ~]# kubectl  get service
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)           AGE
kubernetes   ClusterIP   10.0.0.1      <none>        443/TCP           8d
web          NodePort    10.0.0.138    <none>        18080:30354/TCP   4m33s
[root@k8s-master1 ~]#
```

使用浏览器访问：

任意一个nodeip:30354

- 升级

```
# 把刚部署的nginx版本改为1.18
kubectl set image deployment/web nginx=nginx:1.18
# 可以进入容器内查看nginx版本
# kubectl  exec  -it  podid  bash
# 查看升级状态
kubectl rollout status deployment/web
```

```
[root@k8s-master1 ~]# kubectl rollout status deployment/web
deployment "web" successfully rolled out
[root@k8s-master1 ~]#
```

- 回滚

```
# 查看历史记录
kubectl rollout history deployment/web
# 回滚到最新版本
kubectl rollout undo deployment/web
# 回滚到指定版本
kubectl rollout undo deployment/web --to-revision=2
```

- 删除

```
# 删除deployment
kubectl  delete deploy/web
# 删除service
kubectl  delete svc/web
```

四、资源编排（YAML）

1.YAML说明

YAML 是一种简洁的非标记语言。

Kubernetes使用yaml创建资源对象。

kubectl与yaml部署对比：

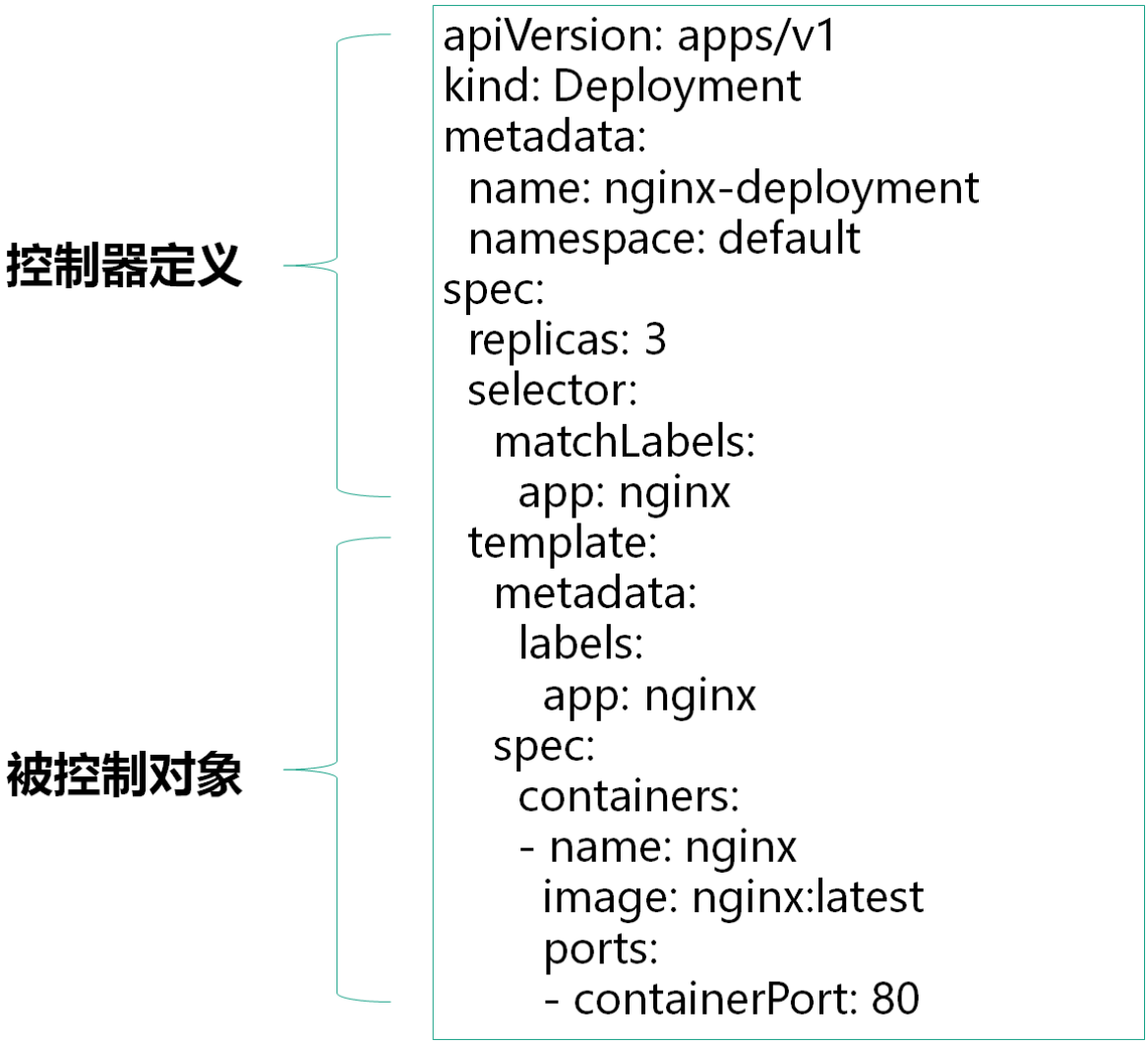
- kubectl适合快速能完成任务
- yaml适合复杂任务
- yaml方面复用

语法格式：

- 缩进表示层级关系
- 不支持制表符“tab”缩进，使用空格缩进
- 通常开头缩进 2 个空格
- 字符后缩进 1 个空格，如冒号、逗号等
- “---” 表示YAML格式，一个文件的开始
- “#”注释

2.YAML内容

在K8S部署应用的YAML文件内容中大致分为两部分：



控制器定义：定义控制器属性

被控制对象：Pod模板，定义容器属性

具体字段意义：

字段名称	意义
apiVersion	API版本
kind	资源类型。
metadata	资源元数据
namespace	命名空间。
spec	资源规格
replicas	副本数量
selector	标签选择器
template	Pod模板
metadata	Pod元数据
spec	Pod规格
containers	容器配置

3.YAML编写技巧

- 获取apiserver版本

  ```
  kubectl   api-versions
  ```

  一般使用控制器部署应用，k8s 1.16版本后使用的都是apps/v1。

  如果不确定api使用的版本，可以去官方文档找对应的示例。

- 用run命令生成部署模板

  ```
  # --dry-run表示生效，只是测试
  kubectl create deployment nginx --image=nginx:1.19 -o yaml --dry-run=client
  > my-deploy.yaml
  ```

- 用get命令将已有部署的应用yaml导出

  ```
  kubectl  get deployment  -o yaml  > test.yaml
  ```

- 如果某个字段内容不记得了，可以通过explain查看更详细的帮助文档获得

  ```
  kubectl explain pod.spec.containers.image
  ```

示例：

- 使用yaml创建一个nginx1.18的deployment

可以参考官网：https://kubernetes.io/zh/docs/tasks/run-application/run-stateless-application-deployment/

```
# yaml内容如下：
# matchLabels /labels中project和app一般对应具体项目名和应用名。也可以自定义添加
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx1.18
spec:
  replicas: 3
  selector:
    matchLabels:
      project: test
      app: nginx1.18
  template:
    metadata:
      labels:
        project: test
        app: nginx1.18
    spec:
      containers:
      - image: nginx:1.18
        name: nginx-web
        ports:
        - containerPort: 80

# 创建
kubectl  apply -f nginx-1.18.yaml

# 查看
kubectl get deploy,pod
```

```
[root@k8s-master1 ~]# cat nginx-1.18.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx1.18
spec:
  replicas: 3
  selector:
    matchLabels:
      project: test
      app: nginx1.18
  template:
    metadata:
      labels:
        project: test
        app: nginx1.18
    spec:
      containers:
      - image: nginx:1.18
        name: nginx-web
        ports:
        - containerPort: 80
[root@k8s-master1 ~]# kubectl  apply -f nginx-1.18.yaml
deployment.apps/nginx1.18 created
```

- 创建一个service

  可以参考官网：https://kubernetes.io/zh/docs/concepts/services-networking/connect-applications-service/

```
# yaml内容如下
apiVersion: v1
kind: Service
metadata:
```

```
  name: nginx-web
  labels:
    project: test
    app: nginx1.18
spec:
  ports:
  - port: 10090
    protocol: TCP
    targetPort: 80
  selector:
    project: test
    app: nginx1.18
  type: NodePort

# 创建
kubectl  apply -f nginx1.18-svc.yaml
```

命令自动部署全：

```
yum install bash-completion -y
source /usr/share/bash-completion/bash_completion
source <(kubectl completion bash)
```

五、Pod详解

1.Pod介绍

- 最小部署单元
- 一组容器的集合
- 一个Pod中的容器共享网络命名空间
- Pod是短暂的

2.Pod存在的意义

Pod为亲密性应用而存在。

亲密性应用场景：

- 两个应用之间发生文件交互
- 两个应用需要通过127.0.0.1或者socket通信
- 两个应用需要发生频繁的调用

3.Pod实现机制与设计模式

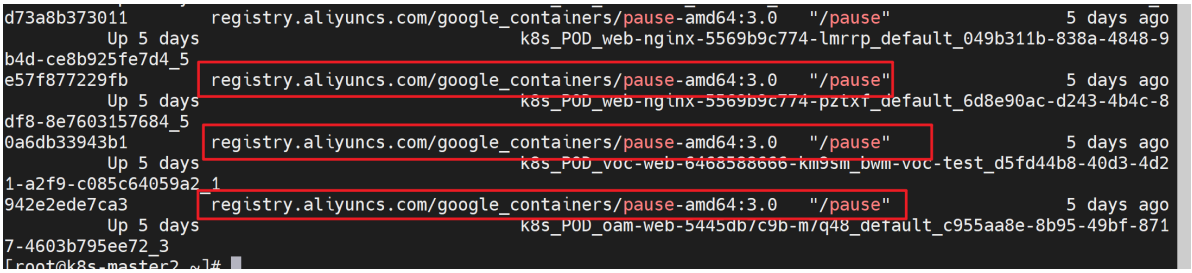
容器之间是通过Namespace隔离的，Pod要想解决上述应用场景，那么就要让Pod里的容器之间高效共享。

具体分为两个部分：网络和存储

- **共享网络**

kubernetes的解法是这样的：会在每个Pod里先启动一个`infra container`小容器（基础容器），然后让其他的容器连接进来这个网络命名空间，然后其他容器看到的网络试图就完全一样了，即网络设备、IP地址、Mac地址等，这就是解决网络共享问题。在Pod的IP地址就是infra container的IP地址。

infra container这个容器使用`kubectl get pod`是看不到的，可以通过`docker ps |grep pause`查看,镜像名叫**pause**



- **共享存储**

比如有两个容器，一个是nginx，另一个是普通的容器，普通容器要想访问nginx里的文件，就需要nginx容器将共享目录通过volume挂载出来，然后让普通容器挂载的这个volume，最后大家看到这个共享目录的内容一样。

示例：

```
# pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: write
    image: centos
    command: ["bash","-c","for i in {1..100};do echo $i >> /data/hello;sleep
1;done"]
    volumeMounts:
      - name: data
        mountPath: /data

  - name: read
    image: centos
    command: ["bash","-c","tail -f /data/hello"]
    volumeMounts:
      - name: data
        mountPath: /data

  volumes:
  - name: data
    emptyDir: {}
```

上述示例中有两个容器，write容器负责提供数据，read消费数据，通过数据卷将写入数据的目录和读取数据的目录都放到了该卷中，这样每个容器都能看到该目录。

验证：

```
kubectl apply -f pod.yaml
kubectl logs my-pod -c read -f
```

在Pod中容器分为以下几个类型：

- **Infrastructure Container**：基础容器，维护整个Pod网络空间，对用户不可见
- **InitContainers**：初始化容器，先于业务容器开始执行，一般用于业务容器的初始化工作

- **Containers**：业务容器，具体跑应用程序的镜像

4.Pod拉取策略

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: nginx
      image: nginx
      imagePullPolicy: IfNotPresent
```

imagePullPolicy 字段有三个可选值：

- IfNotPresent：镜像在宿主机上不存在时才拉取
- Always：每次创建 Pod 都会重新拉取一次镜像(默认值)
- Never： Pod 永远不会主动拉取这个镜像

如果拉取公开的镜像，直接按照上述示例即可，但要拉取私有的镜像，是必须认证镜像仓库才可以，即docker login，而在K8S集群中会有多个Node，显然这种方式是很不放方便的！为了解决这个问题，K8s 实现了自动拉取镜像的功能。 以secret方式保存到K8S中，然后传给kubelet。

5.Pod资源限制

如果节点有足够的可用资源，容器就有可能使用更多资源。我们可以指定容器需要每种资源的数量。要指定的常见资源是CPU和内存。我们为Pod中的容器指定资源请求时，调度程序将决定将Pod放置在哪个节点上。我们为容器指定资源限制时，kubelet会处理限制，以便不允许运行中的容器使用的资源超过我们设置的限制。

Pod资源配额

Pod资源配额有两种：

- 申请配额：调度时使用，参考是否有节点满足该配置

  spec.containers[].resources.requests.cpu

  spec.containers[].resources.requests.memory

- 限制配额：容器能使用的最大配置

  spec.containers[].resources.limits.cpu

  spec.containers[].resources.limits.memory

  requests：容器请求的资源，k8s会根据这个调度到能容纳的Node
  limits：容器最大使用资源

示例：

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: web
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

cpu限制说明：

1核=1000m

1.5核=1500m

那上面限制配置就是1核的二分之一（500m），即该容器最大使用半核CPU。

该值也可以写成浮点数，更容易理解：

半核=0.5

1核=1

1.5核=1.5

6.Pod重启策略

```
apiVersion: v1
kind: Pod
metadata:
  name: web
spec:
  containers:
  - name: nginx
    image: nginx
    restartPolicy: Always
```

restartPolicy字段有三个可选值：

- Always：当容器终止退出后，总是重启容器，默认策略。
- OnFailure：当容器异常退出（退出状态码非0）时，才重启容器。适于job
- Never：当容器终止退出，从不重启容器。适于job

7.Pod健康检查

默认情况下，kubelet 根据容器状态作为健康依据，但不能容器中应用程序状态，例如程序假死。这就会导致无法提供服务，丢失流量。因此引入健康检查机制确保容器健康存活。

7.1健康检查有两种类型：

- livenessProbe (存活探针)
  - 表明容器是否正在运行。
  - 如果存活探测失败，则 kubelet 会杀死容器，并且容器将受到其重启策略影响。
  - 如果容器不提供存活探针，则默认状态为 `Success`。
- readinessProbe （就绪探针）
  - 表明容器是否可以正常接受请求。
  - 如果就绪探测失败，端点控制器将从与 Pod 匹配的所有 Service endpoints中删除该 Pod 的 IP 地址。
  - 初始延迟之前的就绪状态默认为 Failure
  - 如果容器不提供就绪探针，则默认状态为 Success

存活探针适用于希望容器探测失败后被杀死并重新启动，需要指定restartPolicy 为 Always 或 OnFailure。

就绪探针适用于希望Pod在不能正常接收流量的时候被剔除，并且在就绪探针探测成功后才接收流量。

7.2Handler(探针)

探针是kubelet对容器执行定期的诊断，主要通过调用容器配置的三类实现：

- httpGet

  发送HTTP请求，返回200-400范围状态码为成功。

- exec

  执行Shell命令返回状态码是0为成功。

- tcpSocket

  发起TCP Socket建立成功。

7.3 示例

详细的示例参考官网示例：

https://kubernetes.io/zh/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

livenessProbe 的exec进行健康检查示例：

```
apiversion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 60
    livenessProbe:
      exec:
```

```
    command:
    - cat
    - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

initialDelaySeconds：容器启动多久后进行检查

periodSeconds：检查间隔时长，5就是每五秒一次

查看刚启动的pod运行情况：

正常运行，重启次数为0

```
[root@k8s-master1 ~]# kubectl get pod
NAME                     READY   STATUS    RESTARTS   AGE
liveness-exec            1/1     Running   0          7s
nginx1.18-5b4f854b84-2mxdc   1/1     Running   0          4h16m
nginx1.18-5b4f854b84-7fcgs   1/1     Running   0          4h16m
nginx1.18-5b4f854b84-nbp27   1/1     Running   0          4h16m
web-5dcb957ccc-w84f4         1/1     Running   0          7h44m
[root@k8s-master1 ~]#
```

过一段时间查看，pod是正常运行的，但是重启次数已经变为1了

```
[root@k8s-master1 ~]# kubectl get pod
NAME                     READY   STATUS    RESTARTS   AGE
liveness-exec            1/1     Running   1          117s
nginx1.18-5b4f854b84-2mxdc   1/1     Running   0          4h18m
nginx1.18-5b4f854b84-7fcgs   1/1     Running   0          4h18m
nginx1.18-5b4f854b84-nbp27   1/1     Running   0          4h18m
web-5dcb957ccc-w84f4         1/1     Running   0          7h46m
[root@k8s-master1 ~]#
```

上述示例：启动容器第一件事创建文件，停止30s，删除该文件，再停止60s，确保容器还在运行中。

验证现象：容器启动正常，30s后异常，会restartPolicy策略自动重建，容器继续正常，反复现象。

8.Pod调度策略

Pod根据调度器默认算法将Pod分配到合适的节点上，一般是比较空闲的节点。但有些情况我们希望将Pod分配到指定节点，这时我们需要使用调度策略。

三种pod调度策略：nodeName、nodeSelector和污点

1.nodeName

nodeName用于将Pod调度到指定的Node名称上。

示例：

下面示例会绕过调度系统，直接分配到k8s-node1节点。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: busybox
  name: busybox
  namespace: default
spec:
  nodeName: k8s-node1
  containers:
  - image: busybox
```

```
    name: bs
    command:
    - "ping"
    - "baidu.com"
```

nodeName: k8s-node1就是指定node名

2.nodeSelector & nodeAffinity

- nodeSelector

  nodeSelector用于将Pod调度到匹配Label的Node上。先给规划node用途，然后打标签。

例如：集群中有2个node是固态硬盘服务器。现在有个应用对硬盘性能要求高，只想把它部署到固态硬盘的node，那么就可以把这个两个固态硬盘的node打个labe，部署应用时就可以选择对应的labe进行部署。

```
# 给节点打labe
kubectl label nodes k8s-node1 disk=ssd-1
kubectl label nodes k8s-node2 disk=ssd-2

# 查看节点标签
kubectl  get node --show-labels
```

然后在创建Pod只会被调度到含有disk=ssd标签的节点上。

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  nodeSelector:
    disk: ssd
  containers:
  - image: busybox
    name: bs
    command:
    - "ping"
    - "baidu.com"
```

- nodeAffinity

  nodeAffinity节点亲和类似于nodeSelector，可以根据节点上的标签来约束Pod可以调度到哪些节点。

  • 调度分为软策略和硬策略，而不是硬性要求

  - 硬（RequiredDuringSchedulingIgnoredDuringExecution）：必须满足
  - 软（PreferredDuringSchedulingIgnoreDuringExecution）：强调优先满足制定规则，调度器会尝试调度pod到Node上，但并不强求

操作符：

In: label的值在某个列表中
NotIn：label的值不在某个列表中
Exists：某个label存在
DoesNotExist：某个label不存在
Gt：label的值大于某个值（字符串比较）
Lt：label的值小于某个值（字符串比较）

示例：

假设有个node的labe是test，我部署的不希望部署到此节点就使用NotIn

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: tester
            operator: NotIn
            values:
            - test
```

3.污点（taint）与容忍（tolerations）

- 污点

  使用kubectl taint命令可以给某个Node节点设置污点，Node被设置上污点之后就和Pod之间存在了一种相斥的关系，可以让Node拒绝Pod的调度执行，甚至将Node已经存在的Pod驱逐出去

设置污点命令：

```
# 每个污点有一个 key 和 value 作为污点的标签，其中 value 可以为空，effect 描述污点的作用。
kubectl taint node [node] key=value[effect]
```

其中[effect] 可取值：

- NoSchedule ：一定不能被调度。
- PreferNoSchedule：尽量不要调度。
- NoExecute：不仅不会调度，还会驱逐Node上已有的Pod。

示例：

先给节点设置污点，说明这个节点不是谁都可以调度过来的：

```
kubectl taint node k8s-node1  taint_test=123:NoSchedule
# 查看
kubectl describe node k8s-node1 |grep Taints
```

然后在创建Pod只有声明了容忍污点（tolerations），才允许被调度到taint_test=123污点节点上。

```
apiversion: v1
kind: Pod
metadata:
  labels:
    run: busybox
  name: busybox3
```

```
  namespace: default
spec:
  tolerations:
  - key: "taint_test"
    operator: "Equal"
    value: "123"
    effect: "NoSchedule"
  containers:
  - image: busybox
    name: bs
    command:
    - "ping"
    - "baidu.com"
```

如果不配置容忍污点，则永远不会调度到k8s-node1

删除污点：

```
# 格式：kubectl taint node [node] key:[effect]-
kubectl taint node k8s-node1  taint_test-
#或者
kubectl taint node k8s-node1  taint_test=123:NoSchedule-
```

删除其实和添加类似，删除只用在最后面加一个"-"即可

9.故障排查

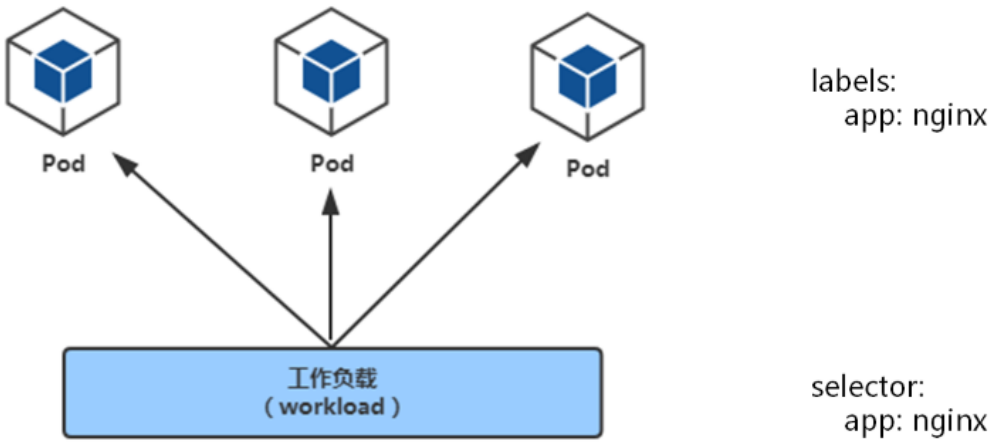
```
# 查看事件，可用于大部分资源
kubectl describe TYPE/NAME
# 如果pod启动失败，先查看日志
kubectl logs TYPE/NAME [-c CONTAINER]
# 进入到容器中debug
kubectl exec POD [-c CONTAINER] -- COMMAND [args...]
```

describe :查看pod事件，创建、启动容器层面的问题
logs:容器日志，观察有没有异常的日志
exec:pod已经启动，进入调试应用

六、常用控制器

1.Pod与controller的关系

- controllers：在集群上管理和运行容器的对象。有时也称为工作负载（workload）
- 通过label-selector相关联，如下图所示。（workload就是指控制器叫法不同）
- Pod通过控制器实现应用的运维，如伸缩，滚动升级等。（单独创建pod是使用这些功能的）



2 .无状态应用-Deployment

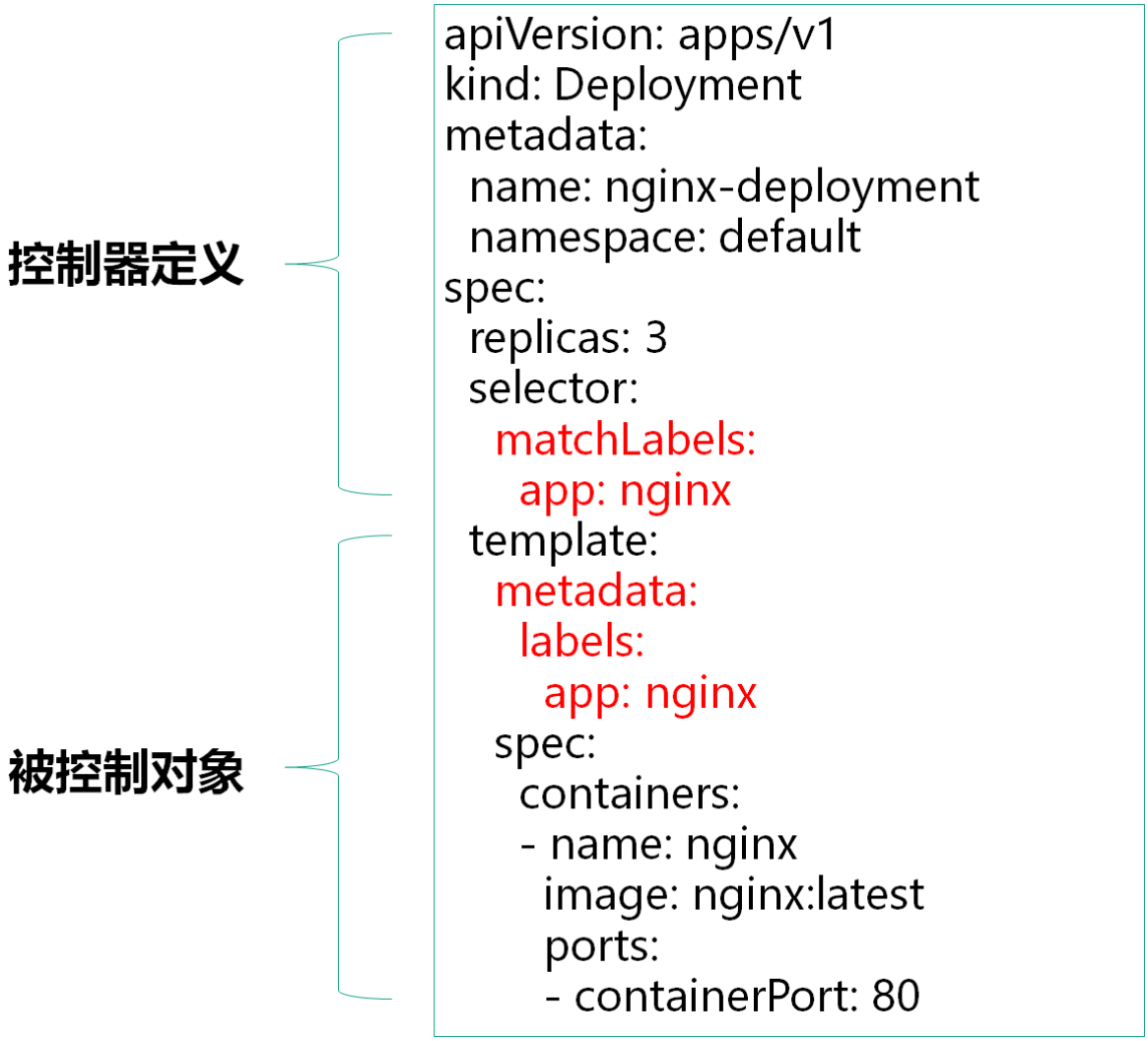
Deployment是最常用的控制器

Deployment功能：

- 部署无状态应用（无状态应用简单来讲，就是Pod可以漂移任意节点，而不用考虑数据和IP变化。）
- 管理Pod和ReplicaSet（副本数量管理控制器）
- 具有上线部署、副本设定、滚动升级、回滚等功能
- 提供声明式更新，例如只更新一个新的Image

应用场景：Web服务，微服务等。

下图是Deployment 标准YAML，通过标签与Pod关联：



示例：

使用YAML部署一个nginx应用：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-nginx
spec:
  replicas: 3    # 设置3个副本
  selector:
    matchLabels:
      app: web-nginx
  template:
    metadata:
      labels:
        app: web-nginx
    spec:
      containers:
      - image: nginx
        name: web-nginx
```

将这个应用暴露到集群外部访问：

```
apiVersion: v1
kind: Service
metadata:
```

```
  labels:
    app: web-nginx
  name: web-nginx
spec:
  ports:
  - port: 80              # 集群内容访问应用端口
    protocol: TCP
    targetPort: 80      # 容器镜像端口
    nodePort: 30008       # 对外暴露的端口
  selector:
    app: web-nginx
  type: NodePort
```

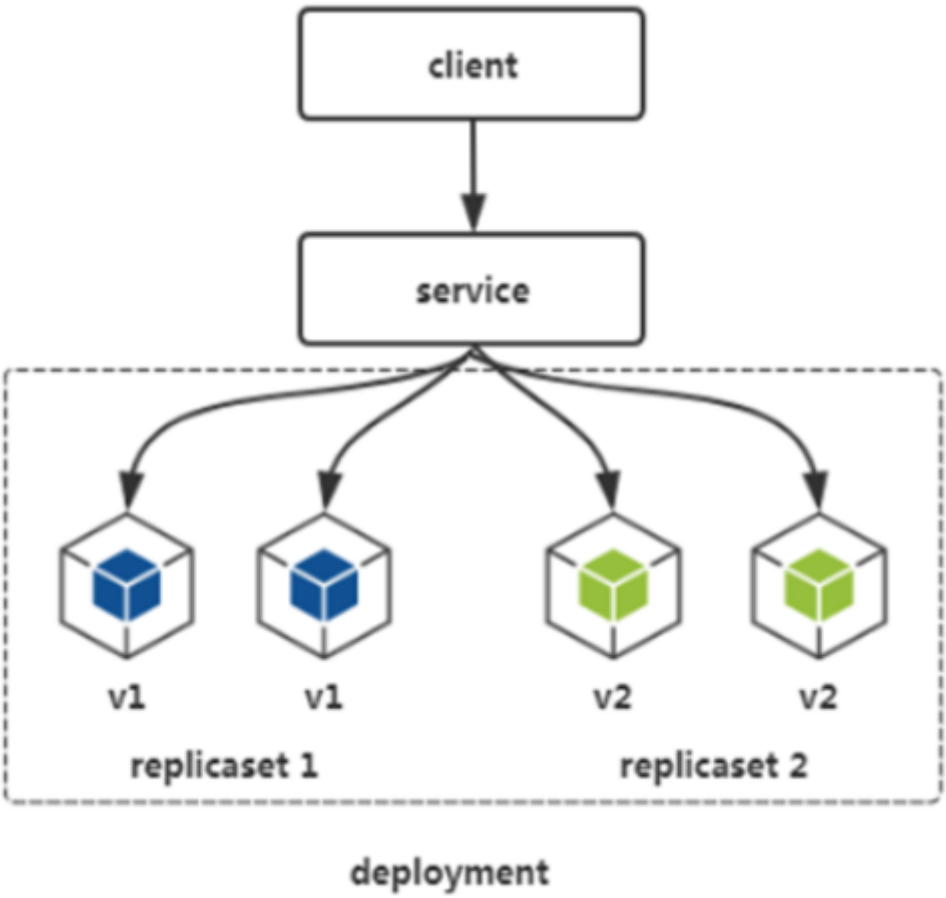
浏览器输入：http://NodeIP:30008 即可访问到该应用。

```
# 升级项目，这里换一个nginx镜像为例：
kubectl set image deployment/web-nginx   web-nginx=nginx:1.17
kubectl rollout status deployment/web-nginx # 查看升级状态

# 如果该版本发布失败想回滚到上一个版本可以执行：
kubectl rollout undo deployment/web-nginx   # 回滚最新版本

# 也可以回滚到指定发布记录：
kubectl rollout history deployment/web-nginx  # 查看发布记录
kubectl rollout undo deployment/web-nginx --to-revision=2  # 回滚指定版本

# 扩容/缩容：
kubectl scale deployment web-nginx --replicas=4
--replicas设置比现在值大就是扩容，反之就是缩容。
```

- 滚动跟新说明

  kubectl set image 会**触发滚动更新**，即分批升级Pod。

  滚动更新原理其实很简单，利用新旧两个replicaset，例如副本是3个，首先Scale Up增加新RS副本数量为1，准备就绪后，Scale Down减少旧RS副本数量为2，以此类推，逐渐替代，最终旧RS副本数量为0，新RS副本数量为3，完成本次更新。这个过程可通过kubectl describe deployment web看到。



3.守护进程-DaemonSet

DaemonSet功能：

- 在每一个Node上运行一个Pod
- 新加入的Node也同样会自动运行一个Pod

应用场景：Agent，例如监控采集工具，日志采集工具

flannel网络插件就是daemonset

```
kubectl  describe pod  kube-flannel-ds-ktrt2 -n kube-system |grep -i DaemonSet
```

```
[root@k8s-master1 ~]# kubectl  describe pod  kube-flannel-ds-ktrt2 -n kube-system |grep -i DaemonSet
Controlled By:  DaemonSet/kube-flannel-ds
[root@k8s-master1 ~]#
[root@k8s-master1 ~]#
[root@k8s-master1 ~]#
```

4.批处理 Job & CronJob

Job：一次性执行

应用场景：离线数据处理，视频解码等业务

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never   # 作业失败后会不再尝试创建新的Pod
  backoffLimit: 4   # .spec.backoffLimit字段限制重试次数。默认情况下，这个字段默认值是6。
```

上述示例中将π计算到2000个位置并将其打印出来。完成大约需要10秒。

查看任务：

```
kubectl get pods,job
```

CronJob：定时任务，像Linux的Crontab一样。

应用场景：通知，备份

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure  # 作业失败并返回状态码非0时，尝试创建新的Pod运行任务
```

上述示例中将每分钟打印一次Hello。

5.有状态应用-StatefulSet

1. StatefulSet说明

上面说的Deployment、DaemonSet都是面向无状态的服务，它们所管理的Pod的IP、名字，启停顺序等都是随机的，而StatefulSet则是有状态服务的应用控制器。什么是有状态服务，一般有一些特殊的标识的服务，如Mysql主从、Es集群等，有启动顺序、数据独立存储等特特殊要求的服务。

- 一般有状态服务有以下特点：

    - 稳定，唯一的网络标识符，持久存储
    - 有序，优雅的部署和扩展、删除和终止
    - 有序，滚动更新
- 有状态应用部署：
    - 考虑实例独立存储
    - 实例之间通信地址要固定
    - 先后启动顺序要考虑

2.稳定的网络

上面说到有状态服务需要唯一的网络标识，这个是通过headless service（无头服务）实现的

一般正常service：

```
apiversion: v1
kind: Service
metadata:
  name: service-test
spec:
  selector:
    app: test
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9999
```

headless service:

```
apiversion: v1
kind: Service
metadata:
  name: service-test
spec:
  clusterIP: None
  selector:
    app: test
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9999
```

普通的service和headless service的区别就是 ClusterIp: None，也就是不指定CluserIP。

如下，使用命令行创建一个headless service，普通的service是有显示虚拟IP，headless service则显示的是None

```
# 创建一个test的deployment
kubectl create deployment testheadless --image=nginx
# 创建一个headless service
kubectl expose deployment testheadless --port=19999 --target-port=80 --cluster-
ip=None --name=testheadless
# 查看
kubectl  get svc |grep Cluster
```

```
[root@k8s-master1 ~]# kubectl  get svc |grep Cluster
kubernetes     ClusterIP   10.0.0.1     <none>        443/TCP          110d
test22         ClusterIP   10.0.0.103   <none>        18099/TCP        16m
testheadless   ClusterIP   None         <none>        19999/TCP        2m
[root@k8s-master1 ~]#
```

- **为什么普通的service需要ClusterIp**：

  无状态的应用Pod是完全一样的，提供相同的服务，可以在飘移在任意节点，例如Web、微服务。而像一些分布式应用程序，例如etcd集群、mysql主从、es集群，每个实例都会维护着一种状态、每个实例都有独立的数据存储，并且每个实例之间必须有固定的访问地址（组建集群），这就是有状态应用。所以有状态应用是不能像无状态应用那样，创建一个标准Service，然后访问ClusterIP负载均衡到一组Pod上。这也是为什么无头Service不需要ClusterIP的原因，它要的是能为每个Pod固定一个网络标识。

- **示例说明**

```yaml
apiVersion: v1
kind: Service
metadata:
  name: headless-svc
spec:
  clusterIP: None
  selector:
    app: nginx-st
  ports:
  - protocol: TCP
    port: 18099
    targetPort: 80

---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web-st
spec:
  selector:
    matchLabels:
      app: nginx-st
  serviceName: "headless-svc"
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx-st
    spec:
      containers:
      - name: nginx-st
        image: nginx
        ports:
        - containerPort: 80
          name: nginx-st
```

StatefulSet和之前的区别就是多了个：`serviceName: "headless-svc"`,这就是告诉StatefulSet要使用`headless-svc`这个headless service来识别pod的身份。

查看：

```
kubectl get statefulset
kubectl get pod |grep web-st
```

```
[root@k8s-master1 ~]# kubectl get statefulset
NAME     READY   AGE
web-st   3/3     13h
[root@k8s-master1 ~]# kubectl get pod |grep web-st
web-st-0                                1/1     Running            0          13h
web-st-1                                1/1     Running            0          13h
web-st-2                                1/1     Running            0          13h
[root@k8s-master1 ~]#
```

- **测试网络**

  创建一个临时的pod来测试dns解析：

```
# 创建临时pod，创建完成后会默认进入pod中
kubectl run -i --tty --image=busybox:1.28.4 dns-test --restart=Never --rm
/bin/sh
# 测试：先测试之前创建的普通的service。规则：nslookup service名.命名空间
nslookup test22.default
```

  可以解析出对应的ip

```
/ # nslookup test22.default
Server:    10.0.0.2
Address 1: 10.0.0.2 kube-dns.kube-system.svc.cluster.local

Name:      test22.default
Address 1: 10.0.0.103 test22.default.svc.cluster.local
/ #
```

```
[root@k8s-master1 ~]# kubectl get svc |grep test22
test22       ClusterIP   10.0.0.103   <none>        18099/TCP        16h
[root@k8s-master1 ~]#
```

```
# 测试headless service
nslookup headless-svc.default
```

  这次解析出来三个pod对应的ip，这里就是**headless service给每个pod一个对应dns名称，保证pod唯一的网络标识符和间通信地址固定。**

```
/ # nslookup headless-svc.default
Server:    10.0.0.2
Address 1: 10.0.0.2 kube-dns.kube-system.svc.cluster.local

Name:      headless-svc.default
Address 1: 10.244.159.134 web-st-1.headless-svc.default.svc.cluster.local
Address 2: 10.244.169.174 web-st-2.headless-svc.default.svc.cluster.local
Address 3: 10.244.169.173 web-st-0.headless-svc.default.svc.cluster.local
/ #
```

- **dns名称规则**

  . ..svc.cluster.local

  示例：web-st-0.nginx.default.svc.cluster.local

3.独立的存储

StatefulSet的存储卷使用VolumeClaimTemplate创建，称为卷申请模板，当StatefulSet使用VolumeClaimTemplate 创建一个PersistentVolume时，同样也会为每个Pod分配并创建一个编号的PVC。

- **示例**

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web-ts2
spec:
  selector:
    matchLabels:
      app: nginx-ts2
  serviceName: 'headless-svc'
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx-ts2
    spec:
      containers:
      - name: nginx-ts2
        image: nginx
        ports:
        - containerPort: 80
          name: nginx-ts2
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "managed-nfs-storage"
      resources:
        requests:
          storage: 1Gi
```

查看pv，pvc。可以看到每个pod有对应编号的pvc

```
[root@k8s-master1 ~]# kubectl  get pv |grep web-ts2
pvc-3b091317-5cc4-491b-9900-5cb84188ba23   1Gi        RWO            Delete           Bound
default/www-web-ts2-1   managed-nfs-storage            5m8s
pvc-b71a4c4b-1375-4a40-938b-7a5f76a57000   1Gi        RWO            Delete           Bound
default/www-web-ts2-2   managed-nfs-storage            4m49s
pvc-ce35b56d-77ec-4331-8620-ac98fca7007a   1Gi        RWO            Delete           Bound
default/www-web-ts2-0   managed-nfs-storage            5m29s
[root@k8s-master1 ~]# kubectl  get pvc |grep web-ts2
www-web-ts2-0   Bound    pvc-ce35b56d-77ec-4331-8620-ac98fca7007a   1Gi        RWO            m
anaged-nfs-storage   5m32s
www-web-ts2-1   Bound    pvc-3b091317-5cc4-491b-9900-5cb84188ba23   1Gi        RWO            m
anaged-nfs-storage   5m11s
www-web-ts2-2   Bound    pvc-b71a4c4b-1375-4a40-938b-7a5f76a57000   1Gi        RWO            m
anaged-nfs-storage   4m52s
[root@k8s-master1 ~]#
```

查看数据目录，也创建了3个对应目录