一、K8s 中的资源

K8s 中所有的内容都抽象为资源， 资源实例化之后，叫做对象

名称空间级别

工作负载型资源(workload)： Pod、ReplicaSet、Deployment、StatefulSet、DaemonSet、Job、CronJob (ReplicationController 在 v1.11 版本被废弃)

服务发现及负载均衡型资源(ServiceDiscovery LoadBalance)： Service、Ingress、...

配置与存储型资源： Volume(存储卷)、CSI(容器存储接口,可以扩展各种各样的第三方存储卷)

特殊类型的存储卷：ConfigMap(当配置中心来使用的资源类型)、Secret(保存敏感数据)、DownwardAPI(把外部环境中的信息输出给容器)

集群级资源：Namespace、Node、Role、ClusterRole、RoleBinding、ClusterRoleBinding

元数据型资源：HPA、PodTemplate、LimitRange

二、资源清单

在 k8s 中，一般使用 yaml 格式的文件来创建符合我们预期期望的 pod ，这样的 yaml 文件我们一般称为资源清单

三、常用字段解释

参数名	字段类型	说明
version	String	这里是指的是K8S API的版本，目前基本上是v1，可以用kubectl api-versions命令查询
kind	String	这里指的是yaml文件定义的资源类型和角色，比如：Pod
metadata	Object	元数据对象，固定值就写metadata
metadata.name	String	元数据对象的名字，这里由我们编写，比如命名Pod的名字
metadata.namespace	String	元数据对象的命名空间，由我们自身定义
Spec	Object	详细定义对象，固定值就写Spec
spec.containers[]	list	这里是Spec对象的容器列表定义，是个列表
spec.containers[].name	String	这里定义容器的名字
spec.containers[].image	String	这里定义要用到的镜像名称

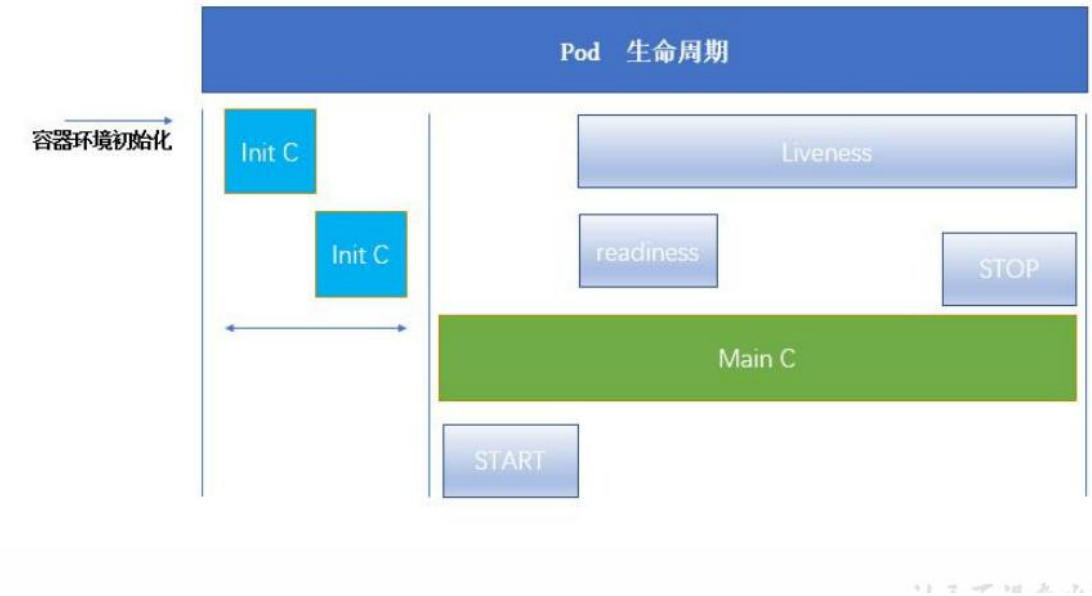
参数名	字段类型	说明
spec.containers[].name	String	定义容器的名字
spec.containers[].image	String	定义要用到的镜像名称
spec.containers[].imagePullPolicy	String	定义镜像拉取策略，有Always、Never、IfNotPresent三个值可选（1）Always：意思是每次都尝试重新拉取镜像 （2）Never：表示仅使用本地镜像 （3）IfNotPresent：如果本地有镜像就使用本地镜像，没有就拉取在线镜像。上面三个值都没设置的话，默认是Always。
spec.containers[].command[]	List	指定容器启动命令，因为是数组可以指定多个，不指定则使用镜像打包时使用的启动命令。
spec.containers[].args[]	List	指定容器启动命令参数，因为是数组可以指定多个。
spec.containers[].workingDir	String	指定容器的工作目录
spec.containers[].volumeMounts[]	List	指定容器内部的存储卷配置
spec.containers[].volumeMounts[].name	String	指定可以被容器挂载的存储卷的名称
spec.containers[].volumeMounts[].mountPath	String	指定可以被容器挂载的存储卷的路径
spec.containers[].volumeMounts[].readOnly	String	设置存储卷路径的读写模式，ture 或者false，默认为读写模式
spec.containers[].ports[]	List	指定容器需要用到的端口列表
spec.containers[].ports[].name	String	指定端口名称
spec.containers[].ports[].containerPort	String	指定容器需要监听的端口号
spec.containers[].ports[].hostPort	String	指定容器所在主机需要监听的端口号，默认跟上面containerPort相同，注意设置了hostPort同一台主机无法启动该容器的相同副本（因为主机的端口号不能相同，这样会冲突）
spec.containers[].ports[].protocol	String	指定端口协议，支持TCP和UDP，默认值为TCP
spec.containers[].env[]	List	指定容器运行前需设置的环境变量列表

spec.containers[].env[].name	String	指定环境变量名称
spec.containers[].env[].value	String	指定环境变量值
spec.containers[].resources	Object	指定资源限制和资源请求的值（这里开始就是设置容器的资源上限）
spec.containers[].resources.limits	Object	指定设置容器运行时资源的运行上限
spec.containers[].resources.limits.cpu	String	指定CPU的限制，单位为core数，将用于docker run --cpu-shares参数（这里前面文章Pod资源限制有讲过）
spec.containers[].resources.limits.memory	String	指定MEM内存的限制，单位为MIB、GiB
spec.containers[].resources.requests	Object	指定容器启动和调度时的限制设置
spec.containers[].resources.requests.cpu	String	CPU请求，单位为core数，容器启动时初始化可用数量
spec.containers[].resources.requests.memory	String	内存请求，单位为MIB、GiB，容器启动的初始化可用数量

参数名	字段类型	说明
spec.restartPolicy	String	定义Pod的重启策略，可选值为Always、OnFailure，默认值为Always。 1.Always：Pod一旦终止运行，则无论容器是如何终止的，kubelet服务都将重启它。 2.OnFailure：只有Pod以非零退出码终止时，kubelet才会重启该容器。如果容器正常结束（退出码为0），则kubelet将不会重启它。 3. Never：Pod终止后，kubelet将退出码报告给Master，不会重启该Pod。
spec.nodeSelector	Object	定义Node的Label过滤标签，以key:value格式指定
spec.imagePullSecrets	Object	定义pull镜像时使用secret名称，以name:secretkey格式指定
spec.hostNetwork	Boolean	定义是否使用主机网络模式，默认值为false。设置true表示使用宿主机网络，不使用docker网桥，同时设置了true将无法在同一台宿主机上启动第二个副本。

四、容器的生命周期

下面这个图很好的说明了在敲完命令 kubectl createy -f pod.yaml 之后经历的过程



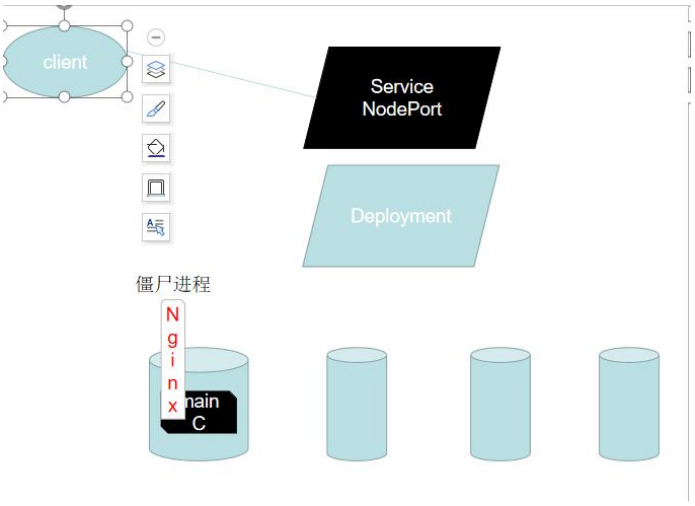
1-

1. init C:初始化容器，初始化完成就会死。可以 0，也可以很多，init C 必须线性运行
2. Pod 一创建就创 Pause
3. MainC 主容器

4.Readinenss：控制器 deployment，下面管理了 4 个 pod，deployment 上创建 nodepoint，可以共享给 k8s 之外，客户端可以访问，

Pod 处在 Running 说明 svc 已经把他放到对外的访问队列，如果还没部署完成主要进程没加载成功，所以需要就绪检测，可用再改成 running

Liveness：main c 可能出现僵尸进程，mainC 还是运行，重启或重建 pod



首先 kubectl 和 api server 交互，然后到 etcd，再到对应 node 的 kubelet

然后 kubelet 和 node 上的 CRI，也就是 docker 交互，开始图中得初始化流程

首先创建好 pod 内容器共享得 pause 网络栈，然后进行一系列得 Init C 流程

初始化工作完成后每个容器都会有自己的 start 初始化脚本要跑，mainc，start 和 stop

可以设定一个探针，在初始化脚本一段时间后检测容器是否可提供服务，如果可以提供服务显示为 running 状态。也就是上面得 readiness 状态 ，只有 readiness 检测成功，pod 才会显示成 running。

同时还有另一个探针，伴随着整个主容器 mainC 的生命周期，不停检测容器是否有异常，并根据 restartPolicy 决定是否要重启。也就是上面得 Liveness 状态

最后在销毁容器之前，还可以运行一段结尾的 stop 脚本

下面我们就用实例对这些流程一个个的深入了解一下。这一节先看看第一个步骤 Init C。

Init C

Init C 是专门用作初始化的容器，其具有如下两个特点：

Pod 能够具有多个容器，应用运行在容器里面，但是它也可能有一个或多个先于应用容器启动的 Init 容器

Init 容器与普通的容器非常像，除了如下两点：

- Init 容器总是运行到成功完成为止
- 每个 Init 容器都必须在下一个 Init 容器启动之前成功完成

如果 Pod 的 Init 容器失败，Kubernetes 会不断地重启该 Pod，直到 Init 容器成功为止。然而，如果 Pod 对应的 restartPolicy 为 Never，它不会重新启动

因为 Init 容器具有与应用程序容器分离的单独镜像，所以它们的启动相关代码具有如下优势：

- 它们可以包含并运行实用工具，但是出于安全考虑，是不建议在应用程序容器镜像中包含这些实用工具的
- 它们可以包含使用工具和定制化代码来安装，但是不能出现在应用程序镜像中。例如，创建镜像没必要 FROM 另一个镜像，只需要在安装过程中使用类似 sed、 awk、 python 或 dig 这样的工具。
- 应用程序镜像可以分离出创建和部署的角色，而没有必要联合它们构建一个单独的镜像。
- Init 容器使用 Linux Namespace，所以相对应用程序容器来说具有不同的文件系统视图。因此，它们能够具有访问 Secret 的权限，而应用程序容器则不能。
- 它们必须在应用程序容器启动之前运行完成，而应用程序容器是并行运行的，所以 Init 容器能够提供了一种简单的阻塞或延迟应用容器的启动的方法，直到满足了一组先决条件。

因为 Init 是区别于业务容器的单独容器，所以其可以完成如下工作：

可以安装某些可以被业务容器使用的工具，这些工具如果封装在业务镜像中会增加业务镜像的冗余
对业务容器的代码进行分离为创建和部署两阶段，将创建阶段放入 Init 容器中
Init 容器属于 Linux 命名空间，相对业务容器具有更高的文件系统权限，例如 Secret 权限。敏感文件处理在 Init 阶段完成使得业务容器安全性更高
针对某些有严重顺序依赖的两个 Pod 来说，可以再需要后启动的 Pod 的 Init 中去做判断，一直等到先启动的 Pod 成功完成，再退出 Init，启动后一个 Pod

Init 不完成，mainc 不开始

pause中实现

- 在 Pod 启动过程中，Init 容器会按顺序在网络和数据卷初始化之后启动。每个容器必须在下一个容器启动之前成功退出
- 如果由于运行时或失败退出，将导致容器启动失败，它会根据 Pod 的 restartPolicy 指定的策略进行重试。然而，如果 Pod 的 restartPolicy 设置为 Always，Init 容器失败时会使用 RestartPolicy 策略
- 在所有的 Init 容器没有成功之前，Pod 将不会变成 Ready 状态。Init 容器的端口将不会在 Service 中进行聚集。 正在初始化中的 Pod 处于 Pending 状态，但应该会将 Initializing 状态设置为 true
- 如果 Pod 重启，所有 Init 容器必须重新执行
- # 对 Init 容器 spec 的修改被限制在容器 image 字段，修改其他字段都不会生效。更改 Init 容器的 image 字段，等价于重启该 Pod

- Init 容器具有应用容器的所有字段。除了 readinessProbe，因为 Init 容器无法定义不同于完成（completion）的就绪（readiness）之外的其他状态。这会在验证过程中强制执行
- 在 Pod 中的每个 app 和 Init 容器的名称必须唯一；与任何其它容器共享同一个名称，会在验证时抛出错误

端口可能有重复，因为init C是串行的

Init 容器实际操作

看 init 本

下面创建一个带 Init 容器的 yaml 配置文件 test-init-main.yaml 如下

busybox 是用在嵌入式中的轻量级 linux 系统，这里用 busybox 做为容器的镜像。containers 下配置了一个主容器，功能只是启动后会打印一条记录，因为是从字符串获取 shell 命令，所以这里要用-c 选项。initContainers 下配置了两个 Init 容器，分别是等待 myservice 和 mydb 这两个服务起来后退出 Init。service 起来后，coreDNS 里面就会有对应的解析 IP，其余 pod 进行域名解析的话会自动获取到

shell 命令的语法这里就不细说了，until do done 的结构是直到 until 条件满足跳出循环

同时选择 image 的时候最好加上一个 tag，而不要用默认的 latest，因为 latest 的镜像每次都会去重新 pull。即使是用的 latest 的 image，也最好在本地打上自己的 tag 避免重复下载

运行 kubectl apply -f test-init-main.yaml 可以看到容器停在了 Init 阶段，并且两个 Init 容器只完成了 0 个

[root@k8s-master k8s-test]# kubectl apply -f test-init-main.yaml
pod/test-init-main created

[root@k8s-master k8s-test]# **kubectl get pod -o wide**

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
curl-6bf6db5c4f-kljp4 1/1 Running 1 40h 10.244.1.2 k8s-node1 <none> <none>
hellok8s 2/2 Running 0 21h 10.244.1.6 k8s-node1 <none> <none>
test-init-main 0/1 Init:0/2 0 12s 10.244.1.8 k8s-node1 <none> <none>

去看一下 pod 的详细信息

[root@k8s-master k8s-test]# **kubectl describe pod test-init-main**

Name: test-init-main
Namespace: default
Priority: 0
Node: k8s-node1/172.29.56.176
Start Time: Thu, 30 Apr 2020 09:44:01 +0800
Labels: app=myapp
version=v1
Annotations: kubectl.kubernetes.io/last-applied-configuration:

{"apiVersion":"v1","kind":"Pod","metadata":{"annotations":{},"labels":{"app":"myapp","version":"v1"},"name":"test-init-main","namespace":"...
Status: Pending
IP: 10.244.1.8
Init Containers:
init-myservice:
Container ID: docker://58d01f2b5edd15b8f62c83386c2efa20bb1e95c9295b9807c258fb03a14c9486

```
    Image:          busybox
    Image ID:
docker-pullable://busybox@sha256:a8cf7ff6367c2afa2a90acd081b484cbded349a7076e7bdf37a05279f276bc12
    Port:           <none>
    Host Port:      <none>
    Command:
      sh
      -c
      until nslookup myservice; do echo waiting for myservice; sleep 2; done;
    State:          Running
      Started:      Thu, 30 Apr 2020 09:44:06 +0800
    Ready:          False
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-hln8x (ro)
  init-mydb:
    Container ID:
    Image:          busybox
    Image ID:
    Port:           <none>
    Host Port:      <none>
    Command:
      sh
      -c
      until nslookup mydb; do echo waiting for mydb; sleep 2; done;
    State:          Waiting
      Reason:       PodInitializing
    Ready:          False
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-hln8x (ro)
Containers:
  my-busybox:
    Container ID:
    Image:          busybox
    Image ID:
    Port:           <none>
    Host Port:      <none>
    Command:
      sh
      -c
      echo Main app is running && sleep 3600
    State:          Waiting
      Reason:       PodInitializing
    Ready:          False
    Restart Count:  0
    Environment:    <none>
```

```
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-hln8x (ro)
Conditions:
  Type              Status
  Initialized       False
  Ready             False
  ContainersReady   False
  PodScheduled      True
Volumes:
  default-token-hln8x:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-hln8x
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
```

可以看到 init-myservice 这个 Init 容器还处于 running 状态，所以无法向下继续。

这时候我们通过 yaml 文件 test-init-myservice.yaml 去创建一个叫 myservice 的服务

```
apiVersion: v1
kind: Service
metadata:
  name: myservice
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 6666
```

这里是服务 IP 的 80 端口对应着容器的 6666 端口，这里先暂时不用管。确保服务正常起来

```
[root@k8s-master k8s-test]# kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP   43h
myservice    ClusterIP   10.107.244.119   <none>        80/TCP    6s
```

再去看 pod 的状态，发现第一个 Init 容器已经退出了

```
[root@k8s-master k8s-test]# kubectl get pod -o wide
NAME                              READY   STATUS     RESTARTS   AGE    IP            NODE        NOMINATED NODE   READINESS GATES
curl-6bf6db5c4f-kljp4             1/1     Running    1          42h    10.244.1.2    k8s-node1   <none>           <none>
hellok8s                          2/2     Running    0          23h    10.244.1.6    k8s-node1   <none>           <none>
test-init-main                    0/1     Init:1/2   0          115m   10.244.1.8    k8s-node1   <none>           <none>
```

同样再通过 yaml 文件 test-init-mydb.yaml 去创建 mydb 服务

```
apiVersion: v1
kind: Service
metadata:
  name: mydb
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 7777
```

同样确保服务已经起来

[root@k8s-master k8s-test]# **kubectl create -f test-init-mydb.yaml**

```
service/mydb created
[root@k8s-master k8s-test]# kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP   44h
mydb         ClusterIP   10.106.146.185   <none>        80/TCP    27s
myservice    ClusterIP   10.107.244.119   <none>        80/TCP    56m
```

再看 pod 的状态，所有 Init 容器都已经退出，主容器处于 running 状态

```
[root@k8s-master k8s-test]# kubectl get pod -o wide
NAME                         READY   STATUS    RESTARTS   AGE    IP           NODE        NOMINATED
NODE   READINESS GATES
curl-6bf6db5c4f-kljp4        1/1     Running   1          43h    10.244.1.2   k8s-node1   <none>
<none>
hellok8s                     2/2     Running   0          24h    10.244.1.6   k8s-node1   <none>
<none>
test-init-main               1/1     Running   0          155m   10.244.1.8   k8s-node1   <none>
<none>
```

查看一下日志，正好是我们想要打印的内容

[root@k8s-master k8s-test]# **kubectl logs test-init-main**

```
Main app is running
```

Init 容器注意事项

下面是 Init 容器的几个注意事项：

Init 容器是在网络和数据卷初始化之后启动，所以 Init 容器并不是 Pod 的第一个容器。但是我们对网络和数据卷的初始化不能做任何操作

Init 容器只能一个接一个串联执行，而不能并行

在所有的 Init 容器成功之前，Pod 将不会变为 ready 状态，也就意味着不会在 service 中进行聚集

如果 pod 重启，所有的 Init 容器都会被重新执行一遍。这也就意味着 Init 容器多次执行的结果不能有差异

Pod 已经在 running 的情况下可以用 kubectl edit pod xxx 去修改 pod 的一些属性。对 Init 容器只有修改 image 字段才会生效，之后会重启该 pod

在 yaml 文件中配置 containers 和 initContainers 的字段几乎一样，除了 Init 容器没有 readinessProbe

在 yaml 文件中配置的主容器和 Init 容器的名字必须要唯一不能重复

探针

探针是由 kubelet 对容器执行的定期诊断。要执行诊断，kubelet 调用由容器实现的 Handler。有三种类型的处理程序：

- ExecAction：在容器内执行指定命令。如果命令退出时返回码为 0 则认为诊断成功。
- TCPSocketAction：对指定端口上的容器的 IP 地址进行 TCP 检查。如果端口打开，则诊断被认为是成功的。
- HTTPGetAction：对指定的端口和路径上的容器的 IP 地址执行 HTTP Get 请求。如果响应的状态码大于等于200 且小于 400，则诊断被认为是成功的

200成功，300跳转

每次探测都将获得以下三种结果之一：

- 成功：容器通过了诊断。
- 失败：容器未通过诊断。
- 未知：诊断失败，因此不会采取任何行动 挂死

livenessProbe：指示容器是否正在运行。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 的影响。如果容器不提供存活探针，则默认状态为 Success

readinessProbe：指示容器是否准备好服务请求。如果就绪探测失败，端点控制器将从与 Pod 匹配的所有 Service 的端点中删除该 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

探针（Probe）是由 kubelet 对容器进行的定期诊断。注意，探针的目标是容器而不是 pod

livenessProbe 如果失败，则 kubelet 会杀死容器，并根据 restartPolicy 决定是否重启容器。对应 pod 的 running 状态以及 restart 次数。

可以看到主容器使用 nginx 镜像，如果访问 pod_ip:80/fake_index.html 的返回码不在 200 到 400 之间则认为容器还没有就绪

创建容器，查看状态

[root@k8s-master k8s-test]# **kubectl create -f test-readiness-httpget.yaml**
pod/test-readiness-httpget created

[root@k8s-master k8s-test]# **kubectl get pod -o wide**

```
NAME                    READY   STATUS    RESTARTS   AGE     IP            NODE        NOMINATED NODE   READINESS GATES
curl-6bf6db5c4f-kljp4   1/1     Running   1          2d2h    10.244.1.2    k8s-node1   <none>           <none>
hellok8s                2/2     Running   0          31h     10.244.1.6    k8s-node1   <none>           <none>
test-init-main          1/1     Running   6          9h      10.244.1.8    k8s-node1   <none>           <none>
test-readiness-httpget  0/1     Running   0          6s      10.244.1.10   k8s-node1   <none>           <none>
```

可以看到 pod 状态虽然是 running，但是其中的容器却还没有显示 ready 状态。进去看一下 pod 的详细信息

[root@k8s-master k8s-test]# **kubectl describe pod test-readiness-httpget**

```
Name:               test-readiness-httpget
Namespace:          default
Priority:           0
Node:               k8s-node1/172.29.56.176
Start Time:         Thu, 30 Apr 2020 18:48:41 +0800
Labels:             app=myapp
                    version=v1
Annotations:        kubectl.kubernetes.io/last-applied-configuration:

{"apiVersion":"v1","kind":"Pod","metadata":{"annotations":{},"labels":{"app":"myapp","version":"v1"},"name":"test-readin
ess-httpget","name...
Status:             Running
IP:                 10.244.1.10
Containers:
  mynginx:
    Container ID:   docker://272b07aca3c7b9900f6fe91d7418c03315d5f079553c4108f20cd36aadc48f65
    Image:          nginx
    Image ID:
docker-pullable://nginx@sha256:86ae264c3f4acb99b2dee4d0098c40cb8c46dcf9e1148f05d3a51c4df6758c12
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Thu, 30 Apr 2020 18:48:42 +0800
    Ready:          False
    Restart Count:  0
    Readiness:      http-get http://:80/fake_index.html delay=1s timeout=1s period=3s #success=1 #failure=3
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-hln8x (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             False
  ContainersReady   False
  PodScheduled      True
```

Volumes:
  default-token-hln8x:
    Type: Secret (a volume populated by a Secret)
    SecretName: default-token-hln8x
    Optional: false
QoS Class: BestEffort
Node-Selectors: <none>
Tolerations: node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type   Reason   Age   From   Message
  ----   ------   ----   ----   -------
  Normal   Scheduled 2m33s   default-scheduler   Successfully assigned default/test-readiness-httpget to k8s-node1
  Normal   Pulled   <invalid>   kubelet, k8s-node1 Container image "nginx" already present on machine
  Normal   Created   <invalid>   kubelet, k8s-node1 Created container mynginx
  Normal   Started   <invalid>   kubelet, k8s-node1 Started container mynginx
  Warning Unhealthy <invalid> (x22 over <invalid>) kubelet, k8s-node1 **Readiness probe failed: HTTP probe failed** with statuscode: 404

在最下面的 Events 中可以看到，因为就绪探针返回了 404，所以失败，进而导致容器不能处于 ready 状态。

这个时候进入 pod 中的容器，人为添加一个目标文件

[root@k8s-master k8s-test]# **kubectl exec test-readiness-httpget -it -- /bin/bash**
root@test-readiness-httpget:/# **echo "hello fake index" > /usr/share/nginx/html/fake_index.html**
root@test-readiness-httpget:/# **exit**
进入容器的命令为 kubectl exec <pod_name> -c <container_name> -it -- <command>，如果 pod 里面只有一个容器的话可以省略容器名。这里就是运行容器内的 bash，并创建了一个目标文件。

再次查看发现容器已经就绪，并且也可以获取到刚才添加的文件内容

[root@k8s-master k8s-test]# **kubectl get pod -o wide**

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
curl-6bf6db5c4f-kljp4	1/1	Running	1	2d2h	10.244.1.2	k8s-node1	<none>	<none>
hellok8s	2/2	Running	0	31h	10.244.1.6	k8s-node1	<none>	<none>
test-init-main	1/1	Running	6	9h	10.244.1.8	k8s-node1	<none>	<none>
test-readiness-httpget	1/1	Running	0	6m18s	10.244.1.10	k8s-node1	<none>	<none>

[root@k8s-master k8s-test]# **curl 10.244.1.10/fake_index.html**
hello fake index
这里还是使用的 nginx 镜像，不同的是这里是通过在容器内执行命令的 exec 方式做为探针。容器启动后会创建文件

/tmp/test 文件，20 秒钟之后删除。而存活探针就是通过检测这个文件是否存在而决定是否探测成功。

启动 pod，并用-w 参数去持续观察 pod 的状态

[root@k8s-master k8s-test]# **kubectl create -f test-liveness-exec.yaml**

pod/test-liveness-exec created

[root@k8s-master k8s-test]# **kubectl get pod -o wide -w**

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
curl-6bf6db5c4f-kljp4 1/1 Running 1 2d3h 10.244.1.2 k8s-node1 <none> <none>
hellok8s 2/2 Running 0 32h 10.244.1.6 k8s-node1 <none> <none>
test-init-main 1/1 Running 7 10h 10.244.1.8 k8s-node1 <none> <none>
test-liveness-exec 1/1 Running 0 7s 10.244.1.11 k8s-node1 <none> <none>
test-liveness-exec 0/1 Completed 0 22s 10.244.1.11 k8s-node1 <none> <none>
test-liveness-exec 1/1 Running 1 23s 10.244.1.11 k8s-node1 <none>

发现在第 22 秒的时候容器因为文件被删除而被认为不能存活然后被踢出并重启，重启以后新的/tmp/test 文件又被创建所以又开始重复上面的重启步骤。

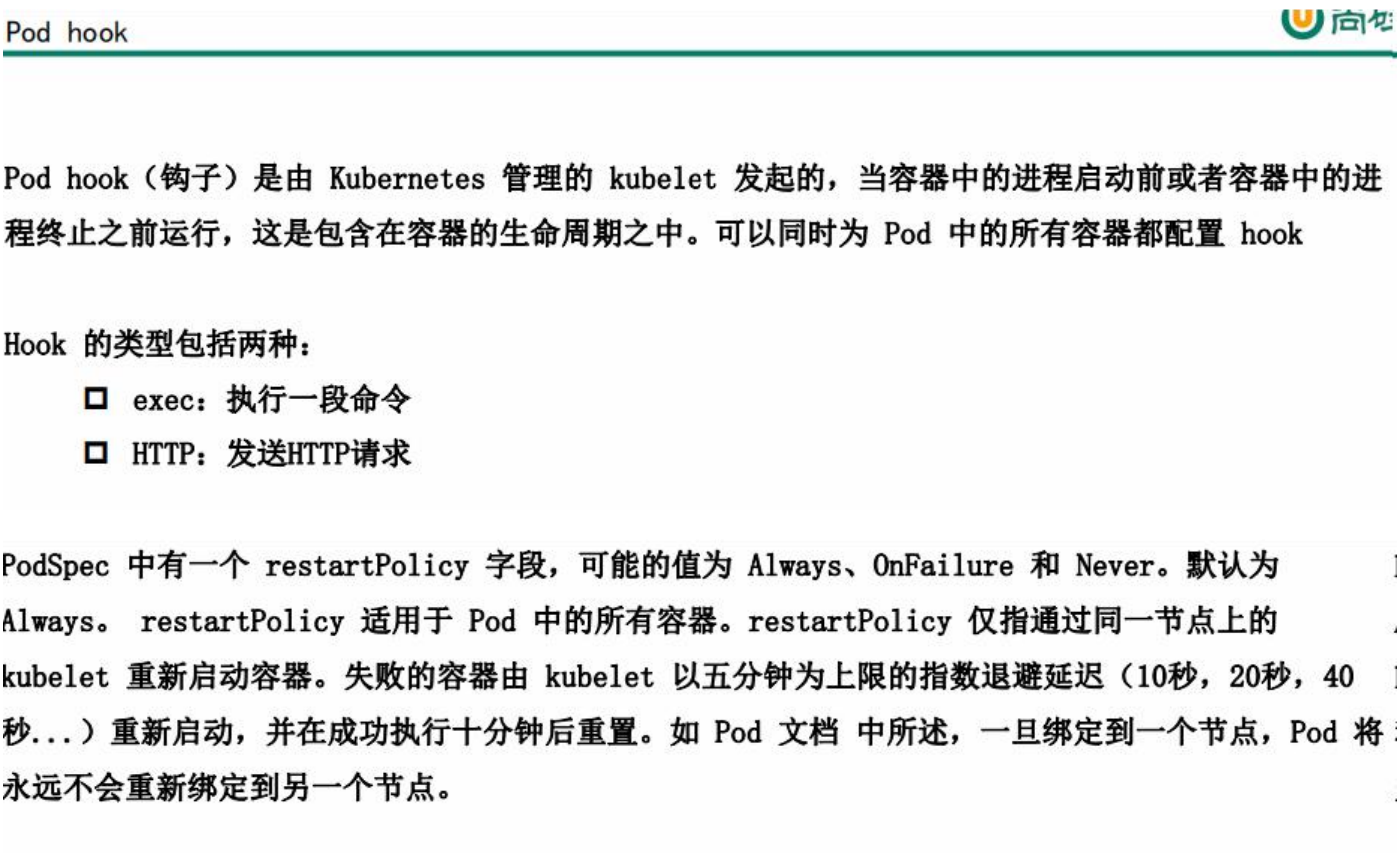
httpget 的方式在上面就绪探针的操作中演示过了这里就不重复了，下面再看看最后一种 tcpsocket 方式的探测方式。

这里会去检测容器的 8080 端口是不是通的，如果超过 3 秒还是没有反馈则表示没有通，探测失败。

启动容器，同样用-w 去持续观察

[root@k8s-master k8s-test]# **kubectl build -f test-liveness-tcpsocket.yaml**
pod/test-liveness-tcpsocket created
[root@k8s-master k8s-test]# **kubectl get pod -o wide -w**
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
curl-6bf6db5c4f-kljp4 1/1 Running 1 2d3h 10.244.1.2 k8s-node1 <none> <none>
hellok8s 2/2 Running 0 32h 10.244.1.6 k8s-node1 <none> <none>
test-init-main 1/1 Running 8 10h 10.244.1.8 k8s-node1 <none> <none>
test-liveness-tcpsocket 1/1 Running 1 13s 10.244.1.12 k8s-node1 <none> <none>
test-liveness-tcpsocket 1/1 Running 2 21s 10.244.1.12 k8s-node1 <none> <none>
test-liveness-tcpsocket 0/1 CrashLoopBackOff 2 30s 10.244.1.12 k8s-node1 <none> <none>
test-liveness-tcpsocket 1/1 Running 3 42s 10.244.1.12 k8s-node1 <none>

发现 pod 隔大约 8 秒重启一次，到第三次触发 CrashLoopBackOff 门限



两种方式混合
当然也可以将 readinessProbe 和 livenessProbe 都配置在容器中彼此独立但共同起作用，这里就不额外演示了，示例这里通过 httpGet 方式来决定容器是否为 ready 状态，而通过检查 8080 端口是否为通的来决定容器是否要重启。

启动和退出动作

顾名思义，就是容器启动完成后和退出之前都可以分别执行一个命令或者脚本去达到某些目的。

通过 yaml 文件 test-start-stop.yaml 来创建一个带启动和退出动作的 pod

这里的例子可能不太具有实际意义，但是不妨碍我们理解原理。容器启动后会往/tmp/start 文件里面写入一句话，结束之前也会在/tmp/stop 文件里面写入一句话。

启动容器，发现成功写入了/tmp/start 文件，但是结束前的那句话这里没法展示了

```
[root@k8s-master k8s-test]# kubectl apply -f test-start-stop.yaml
pod/test-start-stop created
[root@k8s-master k8s-test]# kubectl exec test-start-stop -it -- cat /tmp/start
Hello from postStart handler
```

Pod 的 status

挂起（Pending）：Pod 已被 Kubernetes 系统接受，但有一个或者多个容器镜像尚未创建。等待时间包括调度 Pod 的时间和通过网络下载镜像的时间，这可能需要花点时间

运行中（Running）：该 Pod 已经绑定到了一个节点上，Pod 中所有的容器都已被创建。至少有一个容器正在运行，或者正处于启动或重启状态 running不一定在running

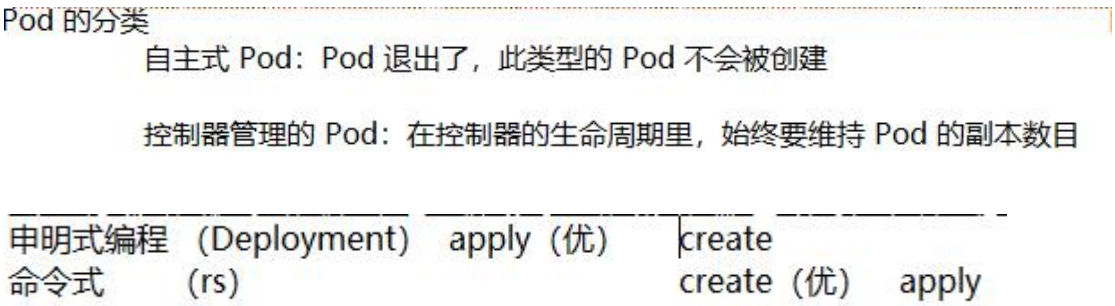
成功（Succeeded）：Pod 中的所有容器都被成功终止，并且不会再重启

失败（Failed）：Pod 中的所有容器都已终止了，并且至少有一个容器是因为失败终止。也就是说，容器以非 0 状态退出或者被系统终止

未知（Unknown）：因为某些原因无法取得 Pod 的状态，通常是因为与 Pod 所在主机通信失败

文章目录

一、控制器类型

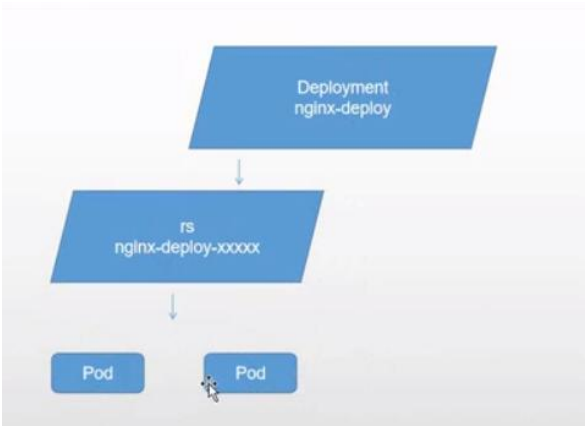


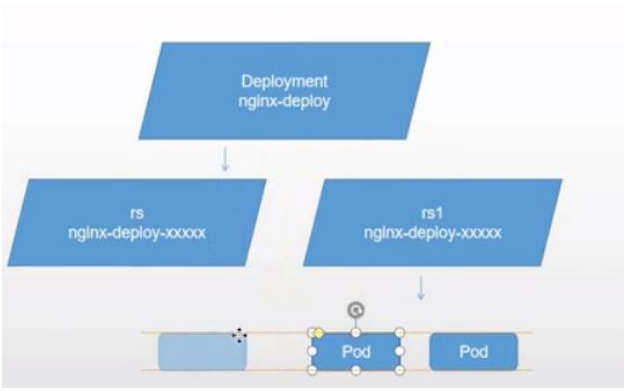
k8s 的控制器一共有如下几种：（纸）

ReplicationController（已弃用）和 ReplicaSet
Deployment
DaemonSet
StatefulSet
Job/CronJob
Horizontal Pod Autoscaling
下面对每种进行详细介绍

ReplicationController 和 ReplicaSet
确保 pod 的副本数量维持在期望水平，但是 RS 可以根据容器的 label 进行集合式的选择
Deployment

Deployment 为 pod 和 RS 提供一个声明式（declarative）方法，替代以前的 RC 来管理应用。





命令式编程：一步步告诉程序应该执行的命令
声明式编程：只声明一个结果，并不给出具体步骤，然后让计算机去实现

典型的应用场景如下：

定义 Deployment 来创建 pod 和 RS
滚动升级和回滚应用
通过创建新的 RS 在下面创建 pod，并停止旧 RS 来进行升级。重新启动旧 RS 而停止新 RS 来达到回滚的目的。
扩容和缩容
暂停和继续 Deployment
在可选 RS 和 Deployment 的情况下，优先选择 Deployment。

DaemonSet
DaemonSet 确保全部或一些 Node 上运行一个 pod 的副本。当有 Node 加入集群时，也会为该 Node 创建一个 pod 副本。

典型的应用场景如下：

运行集群存储的 Daemon，例如 glusterd，ceph
运行日志收集 Daemon，例如 fluentd，logstash

运行监控的 Daemon，例如 Prometheus Node Exporter，collectd 等

Job

在 pod 里面部署一些脚本单次运行，并确保这些 pod 成功结束。

CronJob

类似 linux 的 Cron，用来在给定时间点或者是周期循环执行任务。

StatefulSet

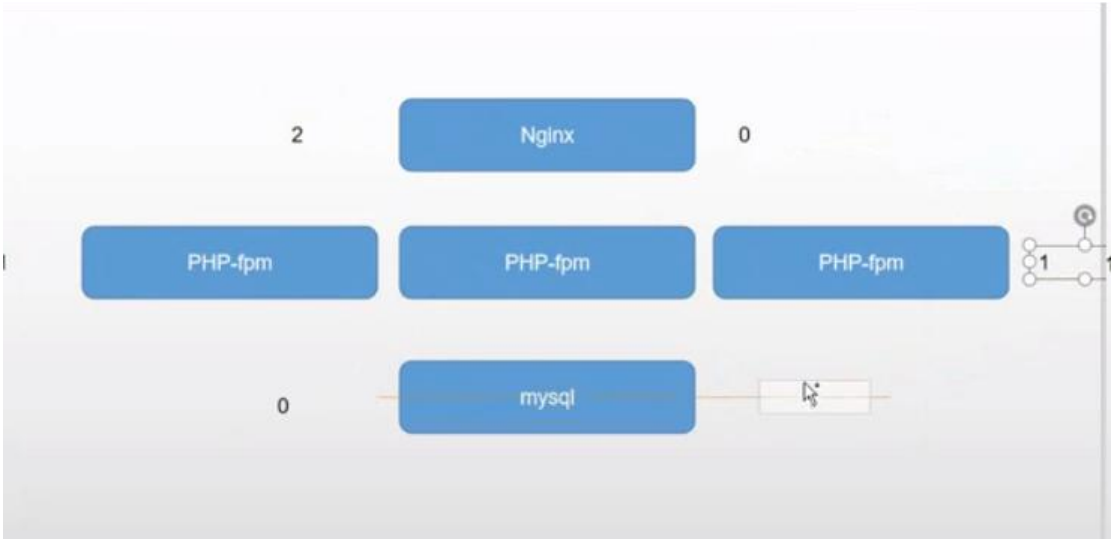
StatefulSet 为 pod 提供了唯一的标识，可以保证部署和 scale 的顺序。区别于 RS 的无状态服务，StatefulSet 是为了解决有状态服务而创建的。典型应用场景如下：

稳定的持久化存储，即 pod 重新调度后还是能访问到完全相同的存储

稳定的网络标识，即 pod 重新调度后其 podname 和 hostname 不变

有序部署，有序扩展，从 0 到 N-1，通过前面说的 Init 容器来实现

有序收缩，有序删除，从 N-1 到 0，后启动的 pod 先停止，避免报错



比如调用 nginx 报错

HPA

HPA 有点像上面这些控制器的附属品，通过一些指标，例如 cpu，去控制之前提到的控制器达到自动扩缩容的目的。HPA 不直接控制 pod。

RS 实际操作

和前面生成 pod 一样，rs 也是通过 yaml 文件来定义。所有字段的详细解释可以通过 kubectl explain rs 或者 kubectl explain ReplicaSet 来查看。

几个重要的字段总结一下

字段　　类型　　说明

apiVersion　　string　　extensions/v1beta1

kind string ReplicaSet
spec object
spec.replicas integer 副本的期望数目，默认为 1
spec.selector object 对 pod 的选择条件
spec.selector.matchLabels object
spec.template object 描述 rs 管理的 pod 的信息
spec.template.metadata object 和 pod 的定义一样，不过注意 labels 内容要和上面的一致
spec.template.spec object 和 pod 的定义一样
补充 apiversion 查询手册 https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.15/

或者通过 kubectl explain xx 来查询 apiversion 怎么填

这里 pod 里面只有一个容器，使用的是我重新 tag 的 nginx 镜像。

在所有可能被分配 pod 的 node 上通过命令 docker tag nginx mynginx:v1 去重新打标签，目的是为了不让 k8s 每次去下载 latest 镜像

创建之后可以看到起了 3 个 pod

[root@k8s-master k8s-test]# **kubectl get pod -o wide**

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
curl-6bf6db5c4f-kljp4 1/1 Running 1 3d18h 10.244.1.2 k8s-node1 <none> <none>
hellok8s 2/2 Running 0 2d23h 10.244.1.6 k8s-node1 <none> <none>
test-init-main 1/1 Running 31 2d1h 10.244.1.8 k8s-node1 <none> <none>
test-rs-74mw2 1/1 Running 0 64m 10.244.1.17 k8s-node1 <none> <none>
test-rs-89l2s 1/1 Running 0 64m 10.244.1.16 k8s-node1 <none> <none>
test-rs-w8zw8 1/1 Running 0 64m 10.244.0.4 k8s-master <none> <none>
test-start-stop 1/1 Running 0 37h 10.244.1.15 k8s-node1
上面一共有 7 个 pod，其中以 test-rs 开头的 3 个 pod 是通过 rs 自动创建的，可以看到其中 2 个在 node1 上，另一个在 master 上。另外 4 个 pod 是之前的实验创建的，它们并不归任何控制器管。

下面我们尝试删除所有的 pod，当然这个命令只会删除 default 命名空间下的 pod，不会删除 kube-system 下的

[root@k8s-master k8s-test]# **kubectl delete pod --all**
pod "curl-6bf6db5c4f-kljp4" deleted
pod "hellok8s" deleted
pod "test-init-main" deleted
pod "test-rs-74mw2" deleted
pod "test-rs-89l2s" deleted
pod "test-rs-w8zw8" deleted
pod "test-start-stop" deleted
接着再查看 pod

[root@k8s-master k8s-test]# **kubectl get pod -o wide**

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
test-rs-gtxln 1/1 Running 0 2m27s 10.244.0.5 k8s-master <none> <none>
test-rs-hn4g5 1/1 Running 0 2m27s 10.244.1.20 k8s-node1 <none> <none>
test-rs-wrrh2 1/1 Running 0 2m27s 10.244.1.19 k8s-node1
发现不是由 RS 创建的 pod 都不会重启，而由 RS 创建的 3 个 pod 又改了个名字重新出现了。这就是自主式 pod 和由控制器控制的 pod 的一个最大不同。

此时如果看一下这三个 pod 的 label

[root@k8s-master k8s-test]# **kubectl get pod -o wide --show-labels**
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES LABELS
test-rs-gtxln 1/1 Running 0 11h 10.244.0.5 k8s-master <none> <none> app=rs-app
test-rs-hn4g5 1/1 Running 0 11h 10.244.1.20 k8s-node1 <none> <none> app=rs-app
test-rs-wrrh2 1/1 Running 0 11h 10.244.1.19 k8s-node1 <none> <none> app=rs-app
这三个 pod 的 label 是一致的，都是 app=rs-app。这时我们修改其中一个 pod 的 label

[root@k8s-master k8s-test]# **kubectl label pod test-rs-wrrh2 --overwrite app=rs-app1**
pod/test-rs-wrrh2 labeled
[root@k8s-master k8s-test]# **kubectl get pod --show-labels**
NAME READY STATUS RESTARTS AGE LABELS
test-rs-hn4g5 1/1 Running 0 11h app=rs-app
test-rs-n5zxb 1/1 Running 0 5s app=rs-app
test-rs-pph4b 1/1 Running 0 101s app=rs-app
test-rs-wrrh2 1/1 Running 0 11h app=rs-app1
通过 kubectl label --help 可以查看标签的一些命令，要修改一个 pod 已经存在的一个标签要

加上**--overwrite**

我们发现 rs 又给我们新建了一个标签为 app=rs-app 的 pod。

这就是 rs 的一个机制，就是通过 label 去判断到底哪些 pod 是归这个 rs 管的。当有一个 pod 的 label 变了，rs 发现自己下面的 pod 少了一个，就又会新建一个来达到期望值 3。而 app=rs-app1 的 pod 就变为了自主 pod，删除了也不会再重启了。

现在删除刚才创建的 rs

```
[root@k8s-master k8s-test]# kubectl delete rs test-rs
replicaset.extensions "test-rs" deleted
[root@k8s-master k8s-test]# kubectl get pod --show-labels
NAME                READY     STATUS        RESTARTS     AGE      LABELS
test-rs-wrrh2       1/1       Running       0            11h      app=rs-app1
```

属于 rs 下面的 3 个 pod 都随着 rs 被删除，而自主式 pod 并没有影响。

Deployment 实际操作

(纸）
下面这个图很好地表现了 deployment 通过新建 rs 来达到滚动更新以及回退的过程
这里的字段和 rs 中的差不多，不过少了 selector，一会我们就能看到为什么不需要这个 selector 了。

创建好 deployment 后会发现它创建了一个 rs，名字为 deployment 的名字加一个 hash 值，并且下面有 3 个 pod，名字为 rs 的名字后面再接一个 hash 值

```
[root@k8s-master k8s-test]# kubectl apply -f test-deployment.yaml --record
deployment.extensions/test-deployment created
[root@k8s-master k8s-test]# kubectl get deployment
NAME                READY     UP-TO-DATE     AVAILABLE     AGE
test-deployment     3/3       3              3             5s
[root@k8s-master k8s-test]# kubectl get rs
NAME                           DESIRED     CURRENT     READY     AGE
test-deployment-d796d98d4      3           3           3         16s
[root@k8s-master k8s-test]# kubectl get pod -o wide
NAME                                READY     STATUS        RESTARTS     AGE      IP
NODE             NOMINATED NODE     READINESS GATES
test-deployment-d796d98d4-kkm25     1/1       Running       0            21s      10.244.1.29
k8s-node1     <none>             <none>
test-deployment-d796d98d4-qknvs     1/1       Running       0            21s      10.244.1.30
k8s-node1     <none>             <none>
```

test-deployment-d796d98d4-v5v85 1/1 Running 0 21s 10.244.1.28 k8s-node1 <none> <none>
如果查看每个 pod 的标签，会发现 deployment 自动为每个 pod 加上了一个叫做 pod-template-xxx 的**标签乱码**

[root@k8s-master k8s-test]# **kubectl get pod --show-labels**
NAME READY STATUS RESTARTS AGE LABELS
test-deployment-d796d98d4-kkm25 1/1 Running 0 12m app=deployment-app,pod-template-hash=d796d98d4
test-deployment-d796d98d4-qknvs 1/1 Running 0 12m app=deployment-app,pod-template-hash=d796d98d4
test-deployment-d796d98d4-v5v85 1/1 Running 0 12m app=deployment-app,pod-template-hash=d796d98d4
所以即使我们不加上自己的 selector，deployment 也会根据这个新的 label 来进行 pod 选择。

扩缩容

下面来试一下将 3 个 pod 副本自动进行扩缩容。

直接一条命令指定扩缩容后新的副本数量即可，格式为

kubectl scale deployment <deployment_name> --replicas=n

例如将刚才的 pod 副本**扩容**为 5 个

[root@k8s-master k8s-test]# **kubectl scale deployment test-deployment --replicas=5**
deployment.extensions/test-deployment scaled
[root@k8s-master k8s-test]# **kubectl get pod -o wide**
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
test-deployment-d796d98d4-kkm25 1/1 Running 0 18m 10.244.1.29 k8s-node1 <none> <none>
test-deployment-d796d98d4-pwz46 1/1 Running 0 13s 10.244.0.8 k8s-master <none> <none>
test-deployment-d796d98d4-qknvs 1/1 Running 0 18m 10.244.1.30 k8s-node1 <none> <none>
test-deployment-d796d98d4-v5v85 1/1 Running 0 18m 10.244.1.28 k8s-node1 <none> <none>
test-deployment-d796d98d4-w2k2w 1/1 Running 0 13s 10.244.1.31 k8s-node1 <none> <none>
再试试缩容到 1 个

[root@k8s-master k8s-test]# **kubectl scale deployment test-deployment --replicas=1**
deployment.extensions/test-deployment scaled

[root@k8s-master k8s-test]# **kubectl get pod -o wide**
NAME READY STATUS RESTARTS AGE IP
NODE NOMINATED NODE READINESS GATES
test-deployment-d796d98d4-pwz46 1/1 Running 0 65s 10.244.0.8
k8s-master <none> <none>

更新镜像

root@k8s-master01:~# **kubectl get rs**

NAME DESIRED CURRENT READY AGE
nginx-deployment-5754944d6c 2 2 2 32m
扩容缩容不会引起 rs 的改变
root@k8s-master01:~# **kubectl set image deployment/nginx-deployment nginx=busybox**
deployment.extensions/nginx-deployment image updated

root@k8s-master01:~# **kubectl get rs**

NAME DESIRED CURRENT READY AGE
nginx-deployment-5754944d6c 1 1 1 34m
nginx-deployment-69df5c954 2 2 0 11s
镜像的修改会促使 rs 创建
root@k8s-master01:~# **kubectl get rs**
NAME DESIRED CURRENT READY AGE
nginx-deployment-5754944d6c 0 0 0 37m
nginx-deployment-69df5c954 2 2 0 2m58s
并且会更新到最新版本

滚动更新和回滚

下面来试一下前面提到的滚动更新。

首先把 pod 的副本数量恢复到 5 个，便于一会儿查看中间过程

[root@k8s-master k8s-test]# **kubectl scale deployment test-deployment --replicas=5**
deployment.extensions/test-deployment scaled
[root@k8s-master k8s-test]# **kubectl get pod -o wide**
NAME READY STATUS RESTARTS AGE IP
NODE NOMINATED NODE READINESS GATES
test-deployment-d796d98d4-9x5c2 1/1 Running 0 2s 10.244.1.34
k8s-node1 <none> <none>

```
test-deployment-d796d98d4-pwz46   1/1   Running   0   2m41s   10.244.0.8
k8s-master   <none>   <none>
test-deployment-d796d98d4-r2dt7   1/1   Running   0   2s   10.244.1.35
k8s-node1   <none>   <none>
test-deployment-d796d98d4-rrlkc   1/1   Running   0   2s   10.244.1.33
k8s-node1   <none>   <none>
test-deployment-d796d98d4-rs4zx   1/1   Running   0   2s   10.244.1.32
k8s-node1   <none>   <none>
```

既然要更新那必须得给容器准备一个新的镜像，创建下面这个 Dockerfile

```
FROM mynginx:v1
RUN echo 'this is mynginx v2' > /usr/share/nginx/html/index.html
```

这里在 v1 基础上修改了 index.html 的内容，这样一会 curl 比较容易验证。然后生成 v2 版本的 mynginx

```
[root@k8s-master k8s-test]# docker build -t mynginx:v2 .
Sending build context to Docker daemon  12.8kB
Step 1/2 : FROM mynginx:v1
 ---> 602e111c06b6
Step 2/2 : RUN echo 'this is mynginx v2' > /usr/share/nginx/html/index.html
 ---> Running in bdba2f09126d
Removing intermediate container bdba2f09126d
 ---> 418ab2e19eb5
Successfully built 418ab2e19eb5
Successfully tagged mynginx:v2
[root@k8s-master k8s-test]# docker images
REPOSITORY                                  TAG                 IMAGE ID
CREATED                  SIZE
mynginx                                     v2                  418ab2e19eb5        5
seconds ago         127MB
mynginx                                     v1                  602e111c06b6        10
days ago            127MB
nginx                                       latest              602e111c06b6        10
days ago            127MB
```

注意这个新的 image 必须要所有的 node 都可以访问到，不然会出现拉取 image 失败的报错。

对 deployment 的更新命令格式如下

```
kubectl set image deployment/<deployment_name> <container_name>=<new_image_name>
1
```

下面将 mynginx 的版本更新为 v2

```
[root@k8s-master k8s-test]#kubectl set image deployment/test-deployment
mynginx=mynginx:v2
```

deployment.extensions/test-deployment image updated

之后马上查看 **pod** 的状态，此时通过 **pod** 的名字可以发现，似乎旧的 **pod** 在被停止，而一个新的 **rs** 在创建新的 **pod**

[root@k8s-master k8s-test]# **kubectl get pod -o wide**

```
NAME                                READY   STATUS        RESTARTS   AGE   IP
NODE          NOMINATED NODE   READINESS GATES
test-deployment-64484c94f7-4xwk4    1/1     Running       0          6s    10.244.0.9
k8s-master    <none>           <none>
test-deployment-64484c94f7-646lx    1/1     Running       0          6s    10.244.1.39
k8s-node1     <none>           <none>
test-deployment-64484c94f7-7rg5q    1/1     Running       0          97s   10.244.1.37
k8s-node1     <none>           <none>
test-deployment-64484c94f7-dbthf    1/1     Running       0          8s    10.244.1.38
k8s-node1     <none>           <none>
test-deployment-64484c94f7-lctwk    1/1     Running       0          97s   10.244.1.36
k8s-node1     <none>           <none>
test-deployment-d796d98d4-pwz46     0/1     Terminating   0          19m   10.244.0.8
k8s-master    <none>           <none>
test-deployment-d796d98d4-rrlkc     0/1     Terminating   0          16m   10.244.1.33
k8s-node1     <none>           <none>
test-deployment-d796d98d4-rs4zx     0/1     Terminating   0          16m   10.244.1.32
k8s-node1     <none>           <none>
```

稍等一会旧的 **pod** 就全部被新的取代

[root@k8s-master k8s-test]# **kubectl get pod -o wide**

```
NAME                                READY   STATUS    RESTARTS   AGE   IP
NODE          NOMINATED NODE   READINESS GATES
test-deployment-64484c94f7-4xwk4    1/1     Running   0          16s   10.244.0.9
k8s-master    <none>           <none>
test-deployment-64484c94f7-646lx    1/1     Running   0          16s   10.244.1.39
k8s-node1     <none>           <none>
test-deployment-64484c94f7-7rg5q    1/1     Running   0          107s  10.244.1.37
k8s-node1     <none>           <none>
test-deployment-64484c94f7-dbthf    1/1     Running   0          18s   10.244.1.38
k8s-node1     <none>           <none>
test-deployment-64484c94f7-lctwk    1/1     Running   0          107s  10.244.1.36
k8s-node1     <none>           <none>
```

看看是不是新的镜像

```
[root@k8s-master k8s-test]# curl 10.244.0.9
this is mynginx v2
```

升级成功，而正如之前的图片上描述的那样，**deployment** 也确实创建了一个新的 **rs**

[root@k8s-master k8s-test]# **kubectl get rs**

```
NAME                          DESIRED   CURRENT   READY   AGE
test-deployment-64484c94f7    5         5         5       9m14s
test-deployment-d796d98d4     0         0         0       45m
```

旧的 rs 并没有消失，这是为了方便做回滚。

回滚的命令格式如下，回滚到上一个版本

kubectl rollout undo deployment/<deployment_name>
1
回滚到 v1 版本的 mynginx

[root@k8s-master k8s-test]# **kubectl rollout undo deployment/test-deployment**

```
deployment.extensions/test-deployment rolled back
```

[root@k8s-master k8s-test]# **kubectl get pod -o wide**

```
NAME                               READY   STATUS        RESTARTS   AGE   IP
NODE          NOMINATED NODE   READINESS GATES
test-deployment-64484c94f7-646lx   0/1     Terminating   0          11m
10.244.1.39   k8s-node1   <none>           <none>
test-deployment-64484c94f7-lctwk   0/1     Terminating   0          12m
10.244.1.36   k8s-node1   <none>           <none>
test-deployment-d796d98d4-4qjpj    1/1     Running       0          7s
10.244.1.43   k8s-node1   <none>           <none>
test-deployment-d796d98d4-5kvz7    1/1     Running       0          10s
10.244.1.41   k8s-node1   <none>           <none>
test-deployment-d796d98d4-g4stw    1/1     Running       0          8s
10.244.0.10   k8s-master  <none>           <none>
test-deployment-d796d98d4-rpv8w    1/1     Running       0          10s
10.244.1.40   k8s-node1   <none>           <none>
test-deployment-d796d98d4-twrbk    1/1     Running       0          8s
10.244.1.42   k8s-node1   <none>           <none>
```

[root@k8s-master k8s-test]# k**ubectl get pod -o wide**

```
NAME                               READY   STATUS    RESTARTS   AGE   IP
NODE          NOMINATED NODE   READINESS GATES
test-deployment-d796d98d4-4qjpj    1/1     Running   0          10s   10.244.1.43
k8s-node1     <none>           <none>
test-deployment-d796d98d4-5kvz7    1/1     Running   0          13s   10.244.1.41
k8s-node1     <none>           <none>
test-deployment-d796d98d4-g4stw    1/1     Running   0          11s   10.244.0.10
k8s-master    <none>           <none>
test-deployment-d796d98d4-rpv8w    1/1     Running   0          13s   10.244.1.40
k8s-node1     <none>           <none>
test-deployment-d796d98d4-twrbk    1/1     Running   0          11s   10.244.1.42
k8s-node1     <none>           <none>
```

值得一提的就是滚动更新的过程中，k8s 会始终保证有期望数量的 pod 在运行，最多不少于期望减一个 pod。

历史版本

上面提到回滚只能回到上一个版本，而如果再次执行回滚操作又会回到当前版本。那么有没有可能回滚到更早的版本呢，当然是可以的。

查看一下滚动更新的历史纪录

[root@k8s-master k8s-test]# **kubectl rollout history deployment/test-deployment**
deployment.extensions/test-deployment
REVISION CHANGE-CAUSE
2 <none>
3 <none>

我们现在在第 3 个 revision，其中第 1 个 revision 是 v1 版本 mynginx，第 2 个 revision 是 v2 版本的 mynginx，第 3 个 revision 是回滚到的 v1 版本的 mynginx。可以通过字段 deployment.spec.revisionHistoryLimit 去设置要保留多少历史记录，默认是保留所有的记录。

想要回滚到某个 revision，可以用命令

这里我就不演示了。但是值得说明的是，即使 k8s 提供了这种 revision 的功能，但是看起来很不清晰，生产环境还是要做好操作记录和配置备份，通过配置文件去回滚。

同时，还可以查询 rollout 的状态

[root@k8s-master k8s-test]# **kubectl set image deployment/test-deployment mynginx=mynginx:v2**
deployment.extensions/test-deployment image updated
[root@k8s-master k8s-test]# **kubectl rollout status deployment/test-deployment**
Waiting for deployment "test-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "test-deployment" rollout to finish: 2 old replicas are pending termination...
Waiting for deployment "test-deployment" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "test-deployment" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "test-deployment" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "test-deployment" rollout to finish: 4 of 5 updated replicas are available...
deployment "test-deployment" successfully rolled out

DaemonSet 实际操作

DaemonSet 和 RS 唯一的区别就是不用指定副本数量，因为默认是每个 node 有且仅有一个副本。

成功创建，发现每个 node 有一个 pod 副本

[root@k8s-master k8s-test]# **vim test-daemonset.yaml**
[root@k8s-master k8s-test]# **kubectl create -f test-daemonset.yaml**
daemonset.extensions/test-daemonset created
[root@k8s-master k8s-test]# **kubectl get pod -o wide**
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
test-daemonset-7qp6p 1/1 Running 0 10s 10.244.0.12 k8s-master <none> <none>
test-daemonset-ml9c7 1/1 Running 0 10s 10.244.1.48 k8s-node1 <none> <none>
root@k8s-master01:~# **kubectl delete pod deamonset-example-57jdc**
pod "deamonset-example-57jdc" deleted
所有 pod 不会在主节点运行，删除 pod 也是在 node01 重建
后面再详细讲如何通过给 node 添加污点，不让 daemonset 在上面创建 pod。

Job 实际操作

这里通过 nodeName 关键字去选择运行该 Job 的目标 node，也可以用 nodeSelector 批量去选择。如果不指定则 k8s 任意选择一个 node 去执行。

成功创建并执行完 job
root@k8s-master01:~# **vim job.yaml**
root@k8s-master01:~# **kubectl create -f job.yaml**
job.batch/pi created
root@k8s-master01:~# **kubectl get pod**
NAME READY STATUS RESTARTS AGE
pi-ph28g 0/1 ContainerCreating 0 6s
root@k8s-master01:~# **kubectl describe pod pi-ph28g**
Name: pi-ph28g
Namespace: default
Priority: 0
Node: k8s-node02/192.168.66.20
Start Time: Tue, 04 Aug 2020 12:20:19 +0800
Labels: controller-uid=ac3e82f2-8449-4866-a5bd-0d029380a8b3
job-name=pi

```
Annotations:        <none>
Status:             Pending
IP:
Controlled By:    Job/pi
Containers:
  pi:
    Container ID:
    Image:          perl
    Image ID:
    Port:           <none>
    Host Port:      <none>
    Command:
      perl
      -Mbignum=bpi
      -wle
      print bpi(2000)
    State:          Waiting
      Reason:       ContainerCreating
    Ready:          False
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-djvlb (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             False
  ContainersReady   False
  PodScheduled      True
Volumes:
  default-token-djvlb:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-djvlb
    Optional:    false
QoS Class:        BestEffort
Node-Selectors:   <none>
Tolerations:      node.kubernetes.io/not-ready:NoExecute for 300s
                  node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age   From                  Message
  ----    ------     ----  ----                  -------
  Normal  Scheduled  51s   default-scheduler     Successfully assigned default/pi-ph28g to
k8s-node02
  Normal  Pulling    51s   kubelet, k8s-node02   Pulling image "perl"
```

最后一行看到他正在拉镜像

root@k8s-master01:~# **tar -zxvf perl.tar.gz**

perl.tar

root@k8s-master01:~# **docker load -i perl.tar**

root@k8s-master01:~# **scp perl.tar root@k8s-node02:/root/**

[root@k8s-master k8s-test]# **kubectl get job**

NAME COMPLETIONS DURATION AGE

test-job 1/1 29s 29s

[root@k8s-master k8s-test]# **kubectl get pod -o wide**

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

test-job-8qvlr 0/1 Completed 0 34s 10.244.0.13 k8s-master

查看 **pod** 的日志可以看到结果

[root@k8s-master k8s-test]# **kubectl logs test-job-8qvlr**

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410270193852110555964462294895493038196442881097566593344612847564823378678316527120190914564856692346034861045432664821339360726024914127372458700660631558817488152092096282925409171536436789259036001133053054882046652138414695194151160943305727036575959195309218611738193261179310511854807446237996274956735188575272489122793818301194912983367336244065664308602139494639522473719070217986094370277053921717629317675238467481846766940513200056812714526356082778577134275778960917363717872146844090122495343014654958537105079227968925892354201995611212902196086403441815981362977477130996051870721134999999837297804995105973173281609631859502445945534690830264252230825334468503526193118817101000313783875288658753320838142061717766914730359825349042875546873115956286388235378759375195778185778053217122680661300192787661119590921642019 9

ConJob 实际操作

ConJob 是用来创建和管理 job 的，在 cronjob.spec.jobTemplate.spec 下有一些字段需要额外说明一下的

completions - 指定期望的成功运行 job 的 pod 数量，默认为 1

parallelism - 指定并行 pod 并发数，默认为 1

activeDeadlineSeconds - 指定任务运行超时时间，单位为秒

同时在 conjob.spec 下也有一些字段要注意

schedule - 同 linux 中的 contab 的写法

jobTemplate - 嵌套上面 Job 的格式

startingDeadlineSeconds - 任务启动超时时间

concurrencyPolicy - 指定当前一个 job 还没执行完时，又有一个 job 需要被执行的的做法

successfulJobsHistoryLimit - 指定保留多少个成功的历史记录，默认为 3 个

实操

创建出来以后等待约 5 分钟，会发现出现了 3 个 job 以及 3 个对应的 pod

root@k8s-master01:~# **vim cronjob.yaml**

root@k8s-master01:~# **kubectl apply -f cronjob.yaml**

[root@k8s-master k8s-test]# **kubectl get job**

```
NAME                           COMPLETIONS   DURATION   AGE
test-conjob-1588577280   1/1           5s         2m34s
test-conjob-1588577340   1/1           6s         94s
test-conjob-1588577400   1/1           5s         34s
```

[root@k8s-master k8s-test]# **kubectl get pod -o wide**

```
NAME                                READY   STATUS      RESTARTS   AGE     IP
NODE          NOMINATED NODE   READINESS GATES
test-conjob-1588577280-9jp6t   0/1     Completed   0          2m37s   10.244.1.60
k8s-node1   <none>           <none>
test-conjob-1588577340-jzhw6   0/1     Completed   0          97s     10.244.1.62
k8s-node1   <none>           <none>
test-conjob-1588577400-95xc7   0/1     Completed   0          37s     10.244.1.64
k8s-node1   <none>           <none>
```

即使继续等待也只会出现 3 个历史记录，由上面说的 successfulJobsHistoryLimit 参数控制

root@k8s-master01:~# **kubectl get cronjob**

```
NAME    SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
hello   */1 * * * *   False     0        47s             51s
```

root@k8s-master01:~# **kubectl get job**

```
NAME                     COMPLETIONS   DURATION   AGE
hello-1596523680   1/1           20s        56s
pi                       1/1           6m3s       148m
```

root@k8s-master01:~# **kubectl delete job pi**

```
job.batch "pi" deleted
```

root@k8s-master01:~# **kubectl delete daemonset --all**

```
daemonset.extensions "deamonset-example" deleted
```

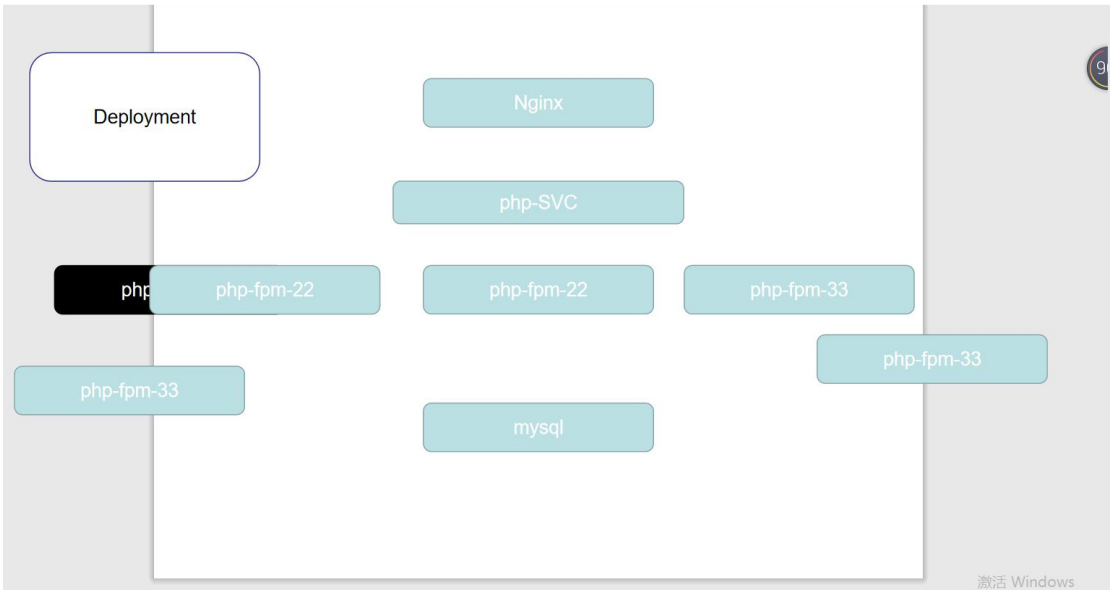
root@k8s-master01:~# **kubectl get pod**

```
NAME                          READY   STATUS             RESTARTS   AGE
frontend-8mrxb                0/1     ErrImagePull       0          4h5m
frontend-8wltp                0/1     ImagePullBackOff   0          4h5m
frontend-94hpz                0/1     ImagePullBackOff   0          4h5m
frontend-lshl8                0/1     ImagePullBackOff   0          4h3m
hello-1596523740-vnjhz   0/1     Completed          0          2m58s
hello-1596523800-fz6lt   0/1     Completed          0          118s
hello-1596523860-9vdjl   0/1     Completed          0          58s
```

root@k8s-master01:~# **kubectl log hello-1596523740-vnjhz**

```
log is DEPRECATED and will be removed in a future version. Use logs instead.
Tue Aug  4 06:49:24 UTC 2020
Hello from the Kubernetes cluster
```

前面我们学习了通过控制器去批量创建和管理 pod，有了 pod 就可以创造一个虚拟 ip 配合负载均衡对外提供服务了。现在新的问题又来了，如何在多个 pod 副本之间形成负载均衡？坏的 pod 被自动替换掉却有了新的 ip，又该如何将新 ip 加入负载均衡？如何将 pod 提供的服务暴露给外网客户端？这些问题都需要 k8s 中一个叫做 Service 的东西来解答，这一节我们就一起来学习下 Service。



.部署一个结构，数据库 mysql，
创建 deployment，通过 deployment 部署 nginx
Pod 可以调用三个 php 达到负载均衡，假设 11 的 pod 死了，deployment 要监控到启动一个新的 pod，意味着其 ip 更改，nginx 去寻找 ip 就发现错误
所以通过 svc
Svc 检测匹配 pod 的信息，加入到 svc 负载队列，pod 有更新会更新到 svc 中
无论是 pod 更新还是扩容都不会对 nginx 或上层服务造成影响

文章目录

Service 的定义

Service 就是 k8s 集群中的一群具有某种共性的 pod，这些 pod 通过 service 的组织对外统一提供服务，service 也被叫做微服务。

每个 service 有自己的一套 label 选择器，符合这些 label 选择器的 pod 就被认为属于该 service，而被该 service 管理。service 会自动获取其管理 pod 的 ip 并制定负载均衡，pod 的 ip 有变化也会自动同步到 service 中。

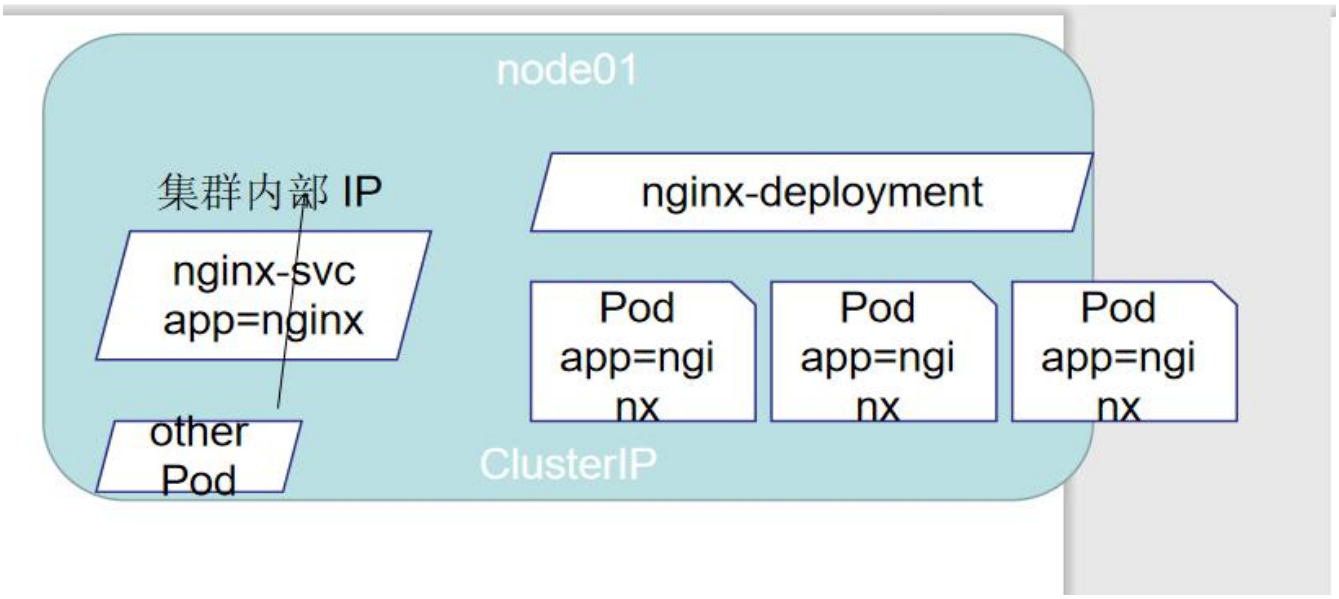
这里说的负载均衡都是基于 IP 的 4 层负载均衡

图中有一个叫 Frontend 的 Service 通过两个标签 app=webapp,role=frontend 去选择符合条件的 pod，只要 pod 的标签中包含这两个标签即认为符合条件，标签可多不可缺。同时 Service 的定义中将 Service 的 IP 的 443 端口映射到后端 pod 的 443 端口。之后外界访问 Service 的 443 端口就会自动负载均衡到后端的 3 个 pod 中的一个。

Service 的类型

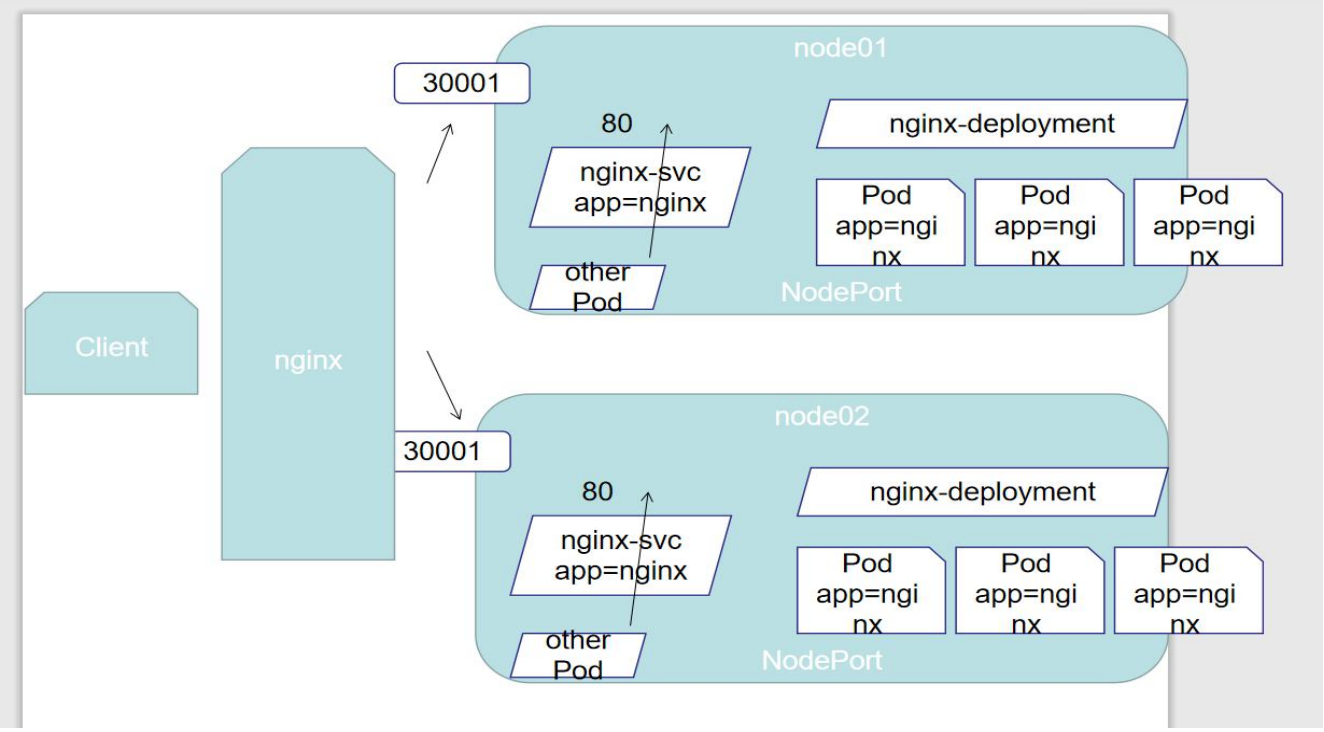
k8s 中的 Service 有如下几种类型，对应着定义 Service 的 yaml 文件的 type 字段

ClusterIP



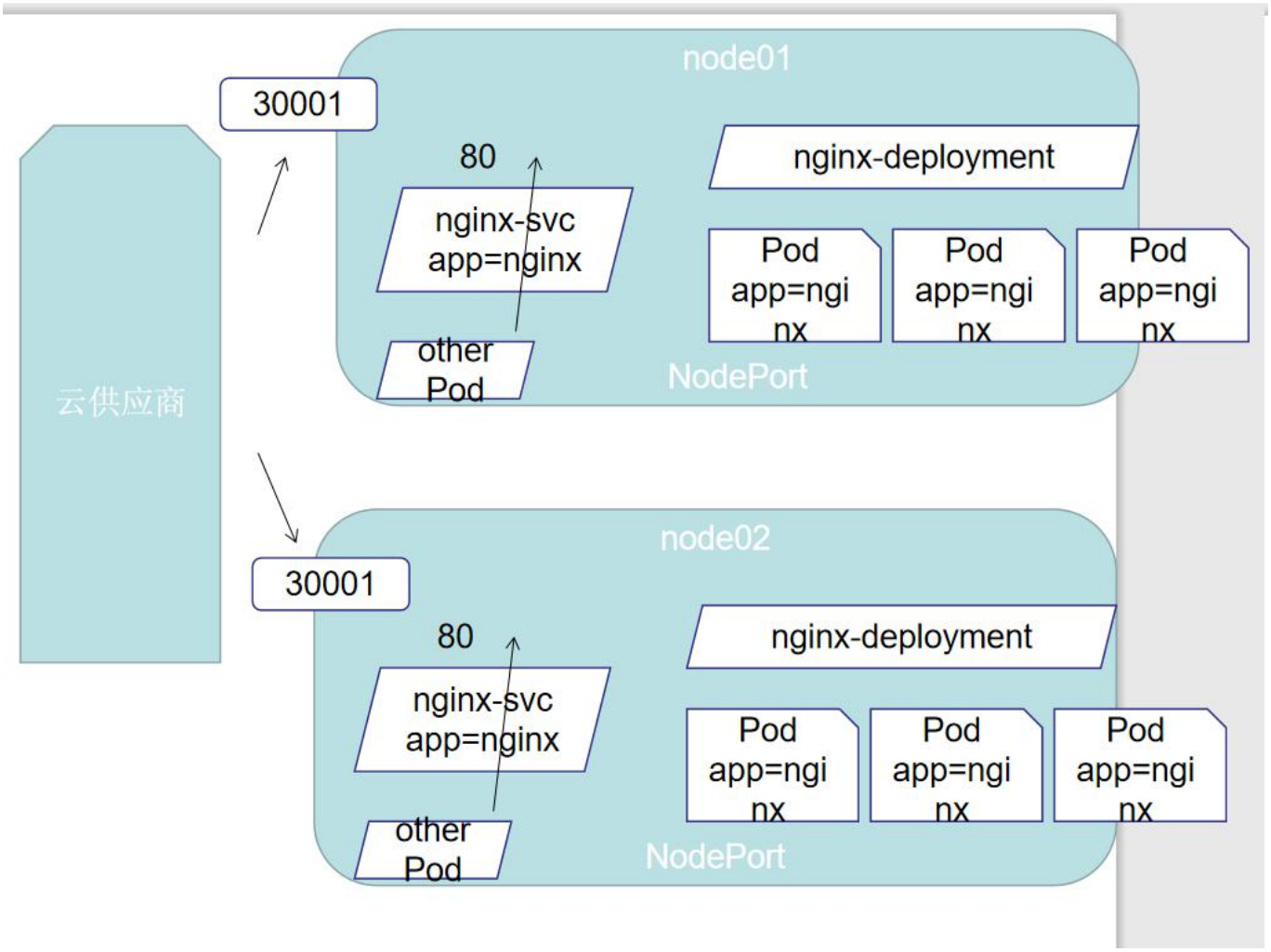
默认类型，**该 Service 的虚拟 IP 仅为集群内部访问,other pod 可以通过访问集群 ip 访问其他三个 pod，只能被集群内部或 node 节点访问**

NodePort



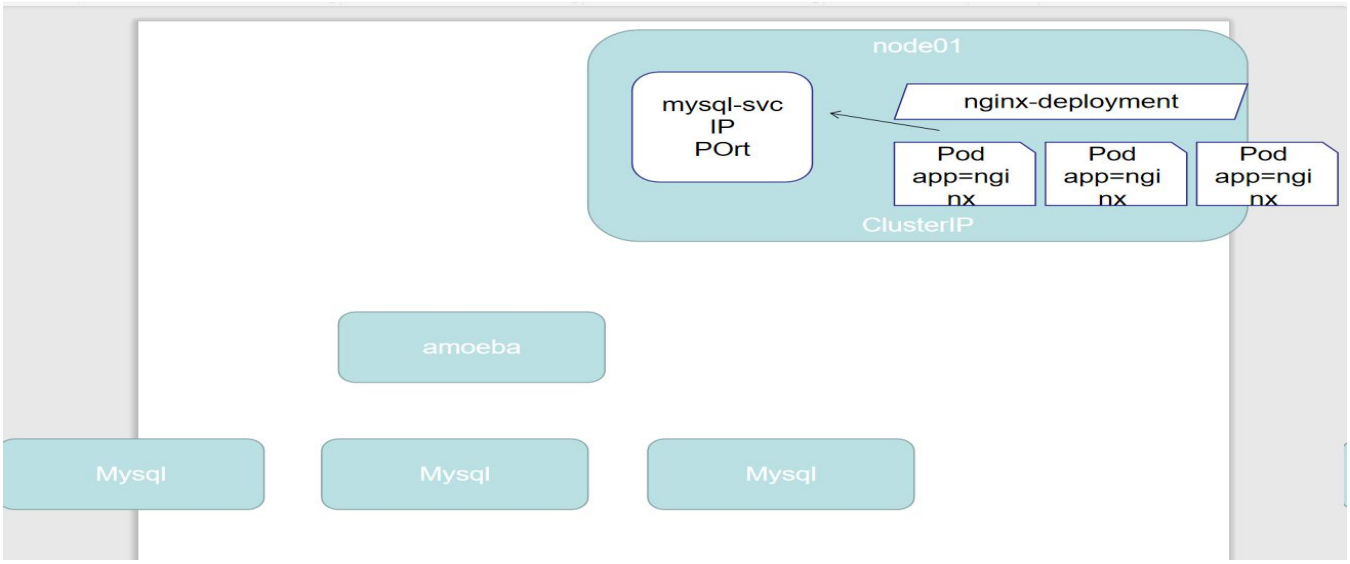
在 **ClusterIP** 的基础上，在每个 Node 的物理网卡上都为该 Service 建立一个相同的端口映射，例如上图中将每个（物理机）Node 的 30001 端口都映射到 Frontend 这个 Service 的 80 端口，相当于访问 3 个 pod，这样不管外部访问哪个 Node 的 IP:30001 都可以访问到该 Service。如果要对集群外提供服务采用该方式，并且通常在 Node 的前面加上针对 Node 物理网卡 IP 的负载均衡

LoadBalancer



在 **NodePort** 的基础上，借助第三方的云服务提供 Node 物理网卡 IP 的负载均衡。企业中用的不多，因为第三方云服务要额外收费，并且完全可以用免费方案代替（与 nodepod 最大区别就是前端的负载均衡控制器不用我们去设置了）

ExternalName



相当于给集群外部的一个第三方服务加了一个 DNS 的 CNAME 记录，将外部流量引入集群内部

Service 实现机制演进

Service 的表象就是每个 Node 的 kube-proxy 进程为 Service 创造了一个虚拟 IP，也就是 VIP，到后端 pod 的映射关系。但是为了达到这个目的，k8s 一直在改进实现机制。

了解这些实现机制，有助于我们去验证和排查 Service 到后端的负载均衡问题。

k8s 1.0 版本，采用 userspace 方式，已经被完全弃用
k8s 1.1 版本，增加了 iptables 方式，并在 k8s 1.2 版本变为默认方式
iptables 会将负载均衡的规则写入 NAT 表中，可以通过 iptabels -nvL -t nat 来查看规则
k8s 1.8 版本，增加了 ipvs 方式，并在 k8s 1.14 版本变为默认方式
ipvs 有自己的管理工具 ipvsadm，可以通过 **ipvsadm -Ln** 来查看规则
不使用 DNS 做负载均衡，因为 DNS 会被缓存，往往达不到均衡的效果

IPVS 因为远高于 iptables 的性能，而在新版本中被优先采用。但是如果没有安装 IPVS 的模块，即使配置了 IPVS 方式，k8s 还是会回退到 iptables 的方式。

修改 iptables 模式为 ipvs 模式
如果因为某些原因发现安装的 1.15 版本的 k8s 还是使用的 iptables 模式，按照以下方式修改为 ipvs 模式。

编辑 kube-proxy 的配置文件

kubectl edit configmap kube-proxy -n kube-system
将其中的 mode ""改为 mode: ipvs，然后:wq 保存退出
删除现有的 kube-proxy，让 k8s 自动替换新的

kubectl delete pod -n kube-system <pod-name>
之后查看新的 kube-poxy 的 log，见到如下字样表示成功启用 ipvs 模式
Using ipvs Proxier
例如

[root@k8s-master k8s-test]# **kubectl logs kube-proxy-4fw7f -n kube-system**
I0506 15:56:08.640887 1 server_others.go:170] Using ipvs Proxier.
W0506 15:56:08.664133 1 proxier.go:401] IPVS scheduler not specified, use rr by default
I0506 15:56:08.664714 1 server.go:534] Version: v1.15.0
I0506 15:56:08.693167 1 conntrack.go:52] Setting nf_conntrack_max to 131072
I0506 15:56:08.693460 1 config.go:96] Starting endpoints config controller
I0506 15:56:08.693485 1 controller_utils.go:1029] Waiting for caches to sync for endpoints config controller
I0506 15:56:08.693596 1 config.go:187] Starting service config controller
I0506 15:56:08.693608 1 controller_utils.go:1029] Waiting for caches to sync for service config controller
I0506 15:56:08.793590 1 controller_utils.go:1036] Caches are synced for endpoints config controller
I0506 15:56:08.793762 1 controller_utils.go:1036] Caches are synced for service config controller

ClusterIP

[root@k8s-master k8s-test]# **kubectl apply -f mynginx-deployment.yaml**
deployment.extensions/mynginx-deployment created
[root@k8s-master k8s-test]# **kubectl get pod -o wide --show-labels**
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES LABELS
mynginx-deployment-b66f59f66-98s7s 1/1 Running 0 11s 10.244.1.70 k8s-node1 <none> <none> app=mynginx,pod-template-hash=b66f59f66,version=v2
mynginx-deployment-b66f59f66-snd7r 1/1 Running 0 11s 10.244.1.69 k8s-node1 <none> <none> app=mynginx,pod-template-hash=b66f59f66,version=v2
mynginx-deployment-b66f59f66-zcp44 1/1 Running 0 11s 10.244.0.14 k8s-master <none> <none> app=mynginx,pod-template-hash=b66f59f66,version=v2 selector:

创建 svc

模拟标签失败
root@k8s-master01:~# **ipvsadm -Ln**
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
-> RemoteAddress:Port Forward Weight ActiveConn InActConn
TCP 10.96.0.1:443 rr
-> 192.168.66.10:6443 Masq 1 3 0
TCP 10.96.0.10:53 rr
-> 10.244.0.13:53 Masq 1 0 0
-> 10.244.0.14:53 Masq 1 0 0
TCP 10.96.0.10:9153 rr
-> 10.244.0.13:9153 Masq 1 0 0
-> 10.244.0.14:9153 Masq 1 0 0

TCP 10.98.77.160:80 rr

TCP 10.102.85.247:80 rr

TCP 10.107.52.95:80 rr

UDP 10.96.0.10:53 rr

  -> 10.244.0.13:53 Masq 1 0 0

  -> 10.244.0.14:53 Masq 1 0 0

错误时会发现其底下没有对应 pod

root@k8s-master01:~# **kubectl delete -f svc.yaml**

service "myapp" deleted

root@k8s-master01:~# **vim svc.yaml**

root@k8s-master01:~# **kubectl apply -f svc.yaml**

service/myapp created

root@k8s-master01:~# **kubectl get svc**

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	47h
myapp	**ClusterIP**	**10.108.12.103**	**<none>**	**80/TCP**	**38s**
mydb	ClusterIP	10.107.52.95	<none>	80/TCP	22h
myservice	ClusterIP	10.98.77.160	<none>	80/TCP	22h

root@k8s-master01:~# **ipvsadm -Ln**

IP Virtual Server version 1.2.1 (size=4096)

Prot LocalAddress:Port Scheduler Flags

  -> RemoteAddress:Port Forward Weight ActiveConn InActConn

TCP 10.98.77.160:80 rr

TCP 10.107.52.95:80 rr

TCP 10.108.12.103:80 rr

  -> 10.244.1.41:80 Masq 1 0 0

  -> 10.244.1.42:80 Masq 1 0 0

  -> 10.244.2.31:80 Masq 1 0 0

UDP 10.96.0.10:53 rr

  -> 10.244.0.13:53 Masq 1 0 0

  -> 10.244.0.14:53 Masq 1 0 0

root@k8s-master01:~# **kubectl get pod -o wide**

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
frontend-lshl8	0/1	ImagePullBackOff	0	5h20m	10.244.1.24	k8s-node02	<none>	<none>
myapp-deploy-6cc7c66999-dx82v	**1/1**	**Running**	**0**	**13m**	**10.244.2.31**	**k8s-node01**	**<none>**	**<none>**
myapp-deploy-6cc7c66999-jlfg4	**1/1**	**Running**	**0**	**13m**	**10.244.1.41**	**k8s-node02**	**<none>**	**<none>**
myapp-deploy-6cc7c66999-jnzj9	**1/1**	**Running**	**0**	**13m**	**10.244.1.42**	k8s-node02		

可以看到 service 已经起来，验证下发现确实是采用 ipvs 方式做的负载均衡

可以看到 Service 的 vip 的 8080 端口后面采用 rr 方式跟了三个 pod 的 ip 的 80 端口，这时为了验证，我事先分别在 3 个 pod 的 index.html 中写入各自的 ip 地址

[root@k8s-master k8s-test]# curl 10.98.205.0:8080

v2 | 10.244.1.70

[root@k8s-master k8s-test]# curl 10.98.205.0:8080
v2 | 10.244.1.69
当然也可以通过域名来访问 service，域名的格式为

<svc_name>.<namespace>.svc.cluster.local

[root@k8s-master k8s-test]# **nslookup mynginx-service.default.svc.cluster.local 10.244.0.2**
Server: 10.244.0.2
Address: 10.244.0.2#53

Name: mynginx-service.default.svc.cluster.local
Address: 10.98.205.0
其中的 10.244.0.2 是其中一个 CoreDNS 的 ip

Headless service

root@k8s-master01:~# **vim svc-none.yaml**
root@k8s-master01:~# **kubectl apply -f svc-none.yaml**
service/myapp-headless created
root@k8s-master01:~# **kubectl get svc**
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
myapp ClusterIP 10.108.12.103 <none> 80/TCP 23m
myapp-headless ClusterIP None <none> 80/TCP 15s
mydb ClusterIP 10.107.52.95 <none> 80/TCP 22h
root@k8s-master01:~# **kubectl get pod -n kube-system**
NAME READY STATUS RESTARTS AGE
coredns-5c98db65d4-bgmvx 1/1 Running 6 47h
coredns-5c98db65d4-zzgjn 1/1 Running 7 47h
etcd-k8s-master01 1/1 Running 5 47h
root@k8s-master01:~# **yum -y install bind-utils**
root@k8s-master01:~# **kubectl get pod -n kube-system -o wide**
NAME READY STATUS RESTARTS AGE IP
NODE NOMINATED NODE READINESS GATES
coredns-5c98db65d4-bgmvx 1/1 Running 6 47h 10.244.0.14
k8s-master01 <none> <none>
coredns-5c98db65d4-zzgjn 1/1 Running 7 47h 10.244.0.13
k8s-master01 <none> <none>
etcd-k8s-master01 1/1 Running 5 47h 192.168.66.10
k8s-master01 <none> <none>
kube-apiserver-k8s-master01 1/1 Running 5 47h 192.168.66.10
k8s-master01 <none> <none>
root@k8s-master01:~# **dig -t A myapp-headless.default.svc.cluster.local. @10.244.0.13**

; <<>> DiG 9.11.4-P2-RedHat-9.11.4-16.P2.el7_8.6 <<>> -t A myapp-headless.default.svc.cluster.local. @10.244.0.13
;; global options: +cmd

;; Got answer:
;; WARNING: .local is reserved for Multicast DNS
;; You are currently testing what happens when an mDNS query is leaked to DNS
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 56964
;; flags: qr aa rd; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;myapp-headless.default.svc.cluster.local. IN A

;; ANSWER SECTION:
myapp-headless.default.svc.cluster.local. 30 IN A 10.244.1.41
myapp-headless.default.svc.cluster.local. 30 IN A 10.244.2.31
myapp-headless.default.svc.cluster.local. 30 IN A 10.244.1.42

;; Query time: 0 msec
;; SERVER: 10.244.0.13#53(10.244.0.13)
;; WHEN: 二 8 月 04 16:42:53 CST 2020
;; MSG SIZE rcvd: 237

NodePort

这里和上面的 ClusterIP 没有太多变化，除了 type 改为了 NodePort，还多了一个 nodePort 指定了一个物理网卡上打开的端口，这个字段也可以不指定，k8s 会自动指定一个。

root@k8s-master01:~# **vim nodeport.yaml**
root@k8s-master01:~# **kubectl apply -f nodeport.yaml**
service/myapp configured
root@k8s-master01:~# **kubectl get pod -o wide**
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
frontend-lshl8 0/1 ErrImagePull 0 6h 10.244.1.24 k8s-node02 <none> <none>
myapp-deploy-6cc7c66999-dx82v 1/1 Running 0 53m 10.244.2.31 k8s-node01 <none> <none>
myapp-deploy-6cc7c66999-jlfg4 1/1 Running 0 53m 10.244.1.41 k8s-node02 <none> <none>
myapp-deploy-6cc7c66999-jnzj9 1/1 Running 0 53m 10.244.1.42 k8s-node02 <none> <none>
root@k8s-master01:~# **kubectl get svc**
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 47h
myapp NodePort 10.108.12.103 <none> 80:30551/TCP 41m
myapp-headless ClusterIP None <none> 80/TCP 18m
mydb ClusterIP 10.107.52.95 <none> 80/TCP 23h

myservice ClusterIP 10.98.77.160 <none> 80/TCP 23h

打开浏览器输入 ip：端口号

root@k8s-master01:~# **netstat -anpt| grep :30551**

tcp6 0 0 :::30551 :::* LISTEN 3523/kube-proxy

root@k8s-master01:~# **ipvsadm -Ln**

IP Virtual Server version 1.2.1 (size=4096)

Prot LocalAddress:Port Scheduler Flags

-> RemoteAddress:Port Forward Weight ActiveConn InActConn

TCP 10.108.12.103:80 rr

-> 10.244.1.41:80 Masq 1 0 0

-> 10.244.1.42:80 Masq 1 0 0

-> 10.244.2.31:80 Masq 1 0 0

TCP 10.244.0.0:30551 rr

-> 10.244.1.41:80 Masq 1 0 0

-> 10.244.1.42:80 Masq 1 0 0

-> 10.244.2.31:80 Masq 1 0 0

TCP 127.0.0.1:30551 rr

-> 10.244.1.41:80 Masq 1 0 0

-> 10.244.1.42:80 Masq 1 0 0

-> 10.244.2.31:80 Masq 1 0 0

TCP 172.17.0.1:30551 rr

-> 10.244.1.41:80 Masq 1 0 0

-> 10.244.1.42:80 Masq 1 0 0

-> 10.244.2.31:80 Masq 1 0 0

TCP 192.168.66.10:30551 rr

-> 10.244.1.41:80 Masq 1 0 0

-> 10.244.1.42:80 Masq 1 0 0

-> 10.244.2.31:80 Masq 1 0 0

TCP 10.107.52.95:80 rr

TCP 10.244.0.1:30551 rr

这时从另外一台集群外的机器上访问两台 node 的 30551 端口就可以看到 pod 的信息，例如从我的 windows 机器上

C:\Users\Admin>curl 172.29.56.175:30000

v2 | 10.244.1.70

C:\Users\Admin>curl 172.29.56.175:30000

v2 | 10.244.1.69

C:\Users\Admin>curl 172.29.56.175:30000

v2 | 10.244.0.14

C:\Users\Admin>curl 172.29.56.176:30000

v2 | 10.244.1.70

C:\Users\Admin>curl 172.29.56.176:30000

v2 | 10.244.1.69

C:\Users\Admin>curl 172.29.56.176:30000

v2 | 10.244.0.14
并且在两台 node 上分别查询 ipvs 信息也会看到

LoadBalancer

因为需要额外收费的第三方云服务，这里就不演示了。相信企业中采用这种方式的也很少。

ExternalName

root@k8s-master01:~# **vim external.yaml**
root@k8s-master01:~# **kubectl apply -f external.yaml**
service/my-service-1 created
root@k8s-master01:~# **kubectl get svc**

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	47h
my-service-1	ExternalName	<none>	hub.atguigu.com	<none>	16s
myapp	NodePort	10.108.12.103	<none>	80:30551/TCP	51m
myapp-headless	ClusterIP	None	<none>	80/TCP	28m
mydb	ClusterIP	10.107.52.95	<none>	80/TCP	23h
myservice	ClusterIP	10.98.77.160	<none>	80/TCP	23h

相当于在内部创建了一个 www.baidu.com 的别名。创建结果如下

这时候在集群内部就可以用 cname 进行查询了，cname 的格式和之前的一样

root@k8s-master01:~# **dig -t A my-service-1.default.svc.cluster.local. @10.244.0.13**

; <<>> DiG 9.11.4-P2-RedHat-9.11.4-16.P2.el7_8.6 <<>> -t A my-service-1.default.svc.cluster.local. @10.244.0.13
;; global options: +cmd
;; Got answer:
;; WARNING: .local is reserved for Multicast DNS
;; You are currently testing what happens when an mDNS query is leaked to DNS
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 46737
;; flags: qr aa rd; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

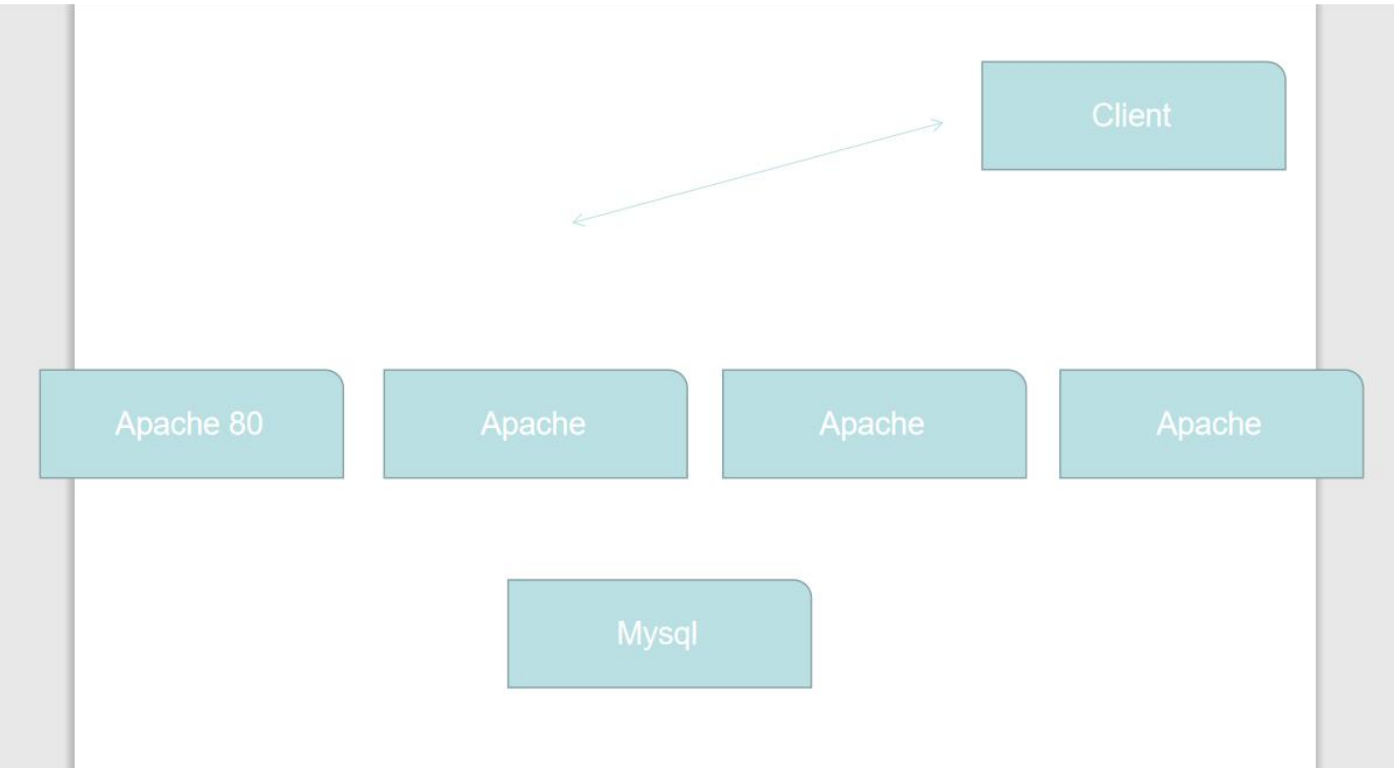
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;my-service-1.default.svc.cluster.local. IN A

;; ANSWER SECTION:
my-service-1.default.svc.cluster.local. 5 IN CNAME hub.atguigu.com.
hub.atguigu.com. 5 IN A 120.240.95.36

;; Query time: 70 msec

;; SERVER: 10.244.0.13#53(10.244.0.13)
;; WHEN: 二 8 月 04 17:02:15 CST 2020
;; MSG SIZE rcvd: 165

Ingress



6-5 讲解

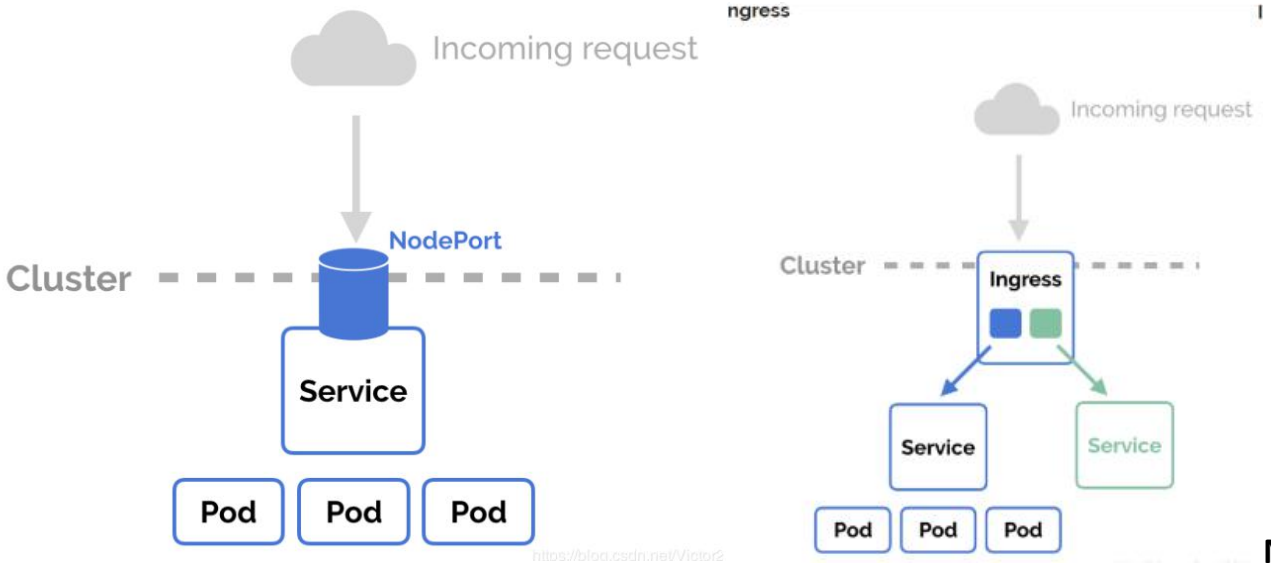
什么是 Ingress
在 k8s 里，Ingress 是一个可以允许集群外部访问集群内布服务的控制器。通过配置一条条规则（rules）来规定进来的连接被分配到后端的哪个服务。

Ingress 相当于一个集中的路由中心，例如，可以将 xiaofu.com/api/v1/路由到后端的 service-v1 服务，而将 xiaofu.com/api/v2/路由到 service-v2 服务。

Ingress vs NodePort

同样是将集群内部的服务暴露给集群外，很有必要对比下 Ingress 和上一节学习的 NodePort。

NodePort



Node

Port 是 Service 的一个类型，并没有额外加入组件。其会在每个 Node 上开一个端口，对应到后端的 Service。所有访问集群任意节点 IP 上该端口都可以访问到后端的 Service。这样的优点就是简单快速，但是只是起了简单的 Node 端口到 Service 端口的映射功能，功能很单一。

Ingress 是区别于上一节学的那些 Service 的另一个单独组件，可以被单独声明，创建和销毁。引入 Ingress 的好处就是有一个集中的路由配置项，同时可实现基于内容的七层负载均衡。小小的麻烦就是引入了额外组件，但是其实创建一个 Ingress 和创建一个 k8s 中的其他资源的步骤没什么两样。

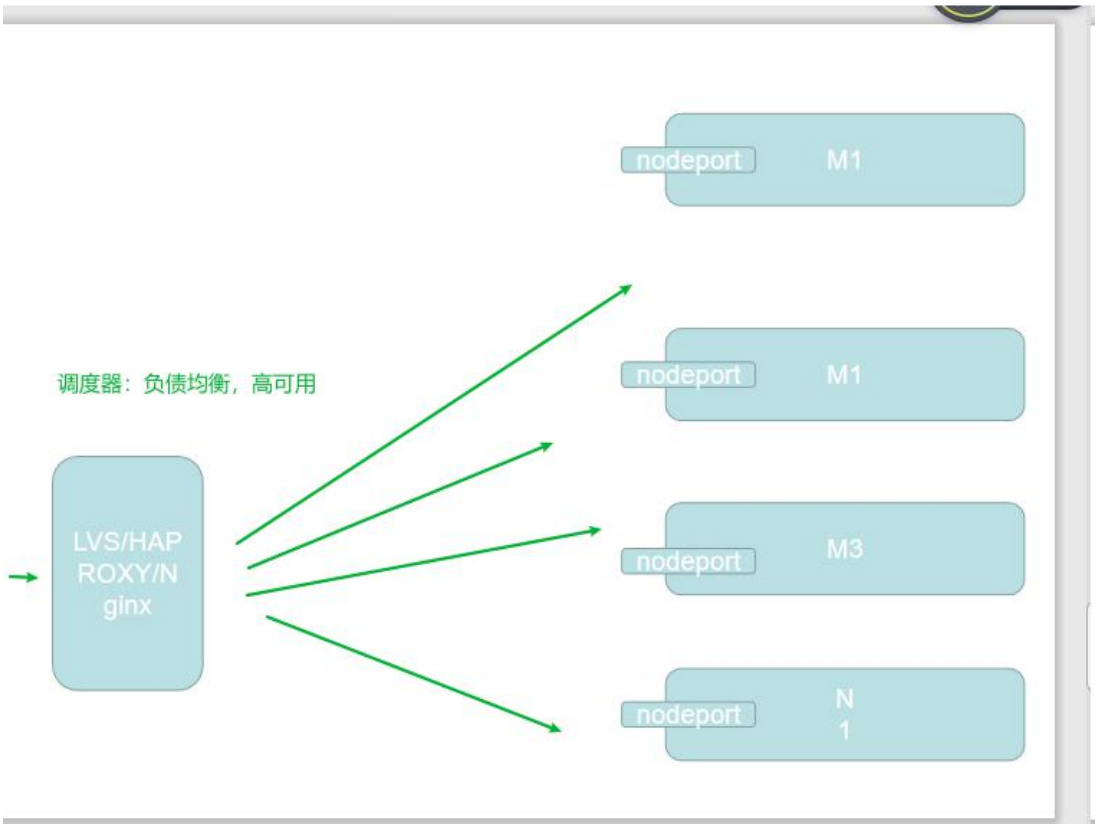
Ingress 的实现方式有很多，下面着重学习下官方推荐的基于 Nginx 的 Ingress-Nginx 控制器。

Ingress-Nginx

可以将上面的 Ingress 理解为在集群的前端加了一个 Nginx，用来提供 Nginx 所能提供的一切功能，例如基于 url 的反向代理，https 认证，用户鉴权，域名重定向等等
3-nginx-ingress.png

需要注意几点：

Nginx 是基于 url 的，如果没有 DNS 解析 url 到集群中任意 ip 可以考虑 hosts 文件劫持
Nginx 的本质也是 NodePort 的方式暴露给集群外的一个特殊 Service
Nginx 的配置文件根据 Ingress 的规则自动添加和修改
更详细的介绍可以查看官方网站。



操作

[root@k8s-node01 ~]# **vim /etc/hosts**

151.101.76.133 raw.githubusercontent.com

[root@k8s-node01 ~]# **wget https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.28.0/deploy/static/mandatory.yaml**

--2020-08-05 13:30:54--
https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.28.0/deploy/static/mandatory.yaml
正在解析主机 raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.76.133
正在连接 raw.githubusercontent.com (raw.githubusercontent.com)|151.101.76.133|:443... 已连接。
已发出 HTTP 请求，正在等待回应... 200 OK
长度：6648 (6.5K) [text/plain]
正在保存至: “mandatory.yaml”

100%[==========>] 6,648 --.-K/s 用时 0.001s

2020-08-05 13:30:56 (9.28 MB/s) - 已保存 “mandatory.yaml” [6648/6648])

[root@k8s-master01 ingress]# **vim service-nodeport.yaml**
拷贝 https://github.com/kubernetes/ingress-nginx/tree/nginx-0.25.1/deploy
[root@k8s-master01 ingress]# **kubectl apply -f service-nodeport.yaml**
service/ingress-nginx created
[root@k8s-master01 ingress]# **kubectl get svc -n ingress-nginx**

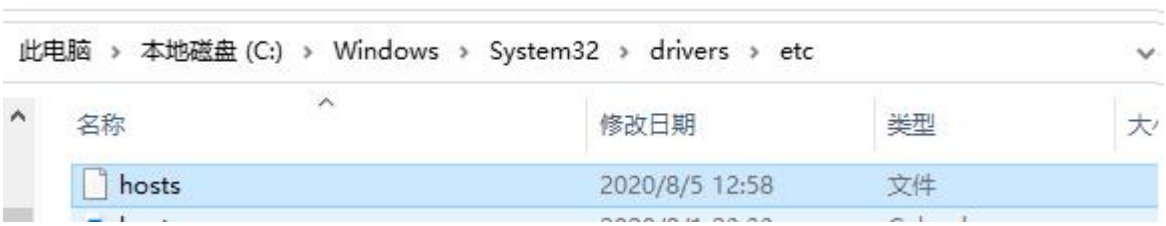
```
NAME                                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
default-http-backend                 ClusterIP   10.97.132.150   <none>        80/TCP
13h
ingress-nginx-controller             NodePort    10.97.38.110    <none>        80:32488/TCP,443:31176/TCP
13h
ingress-nginx-controller-admission   ClusterIP   10.97.202.183   <none>        443/TCP
13h
[root@k8s-master01 ~]# vim ingress-http.yaml
```

见纸前两段

```
[root@k8s-master01 ~]# kubectl apply -f ingress-http.yaml
deployment.extensions/nginx-dm unchanged
service/nginx-svc created
ingress.extensions/nginx-test created

[root@k8s-master01 ~]# kubectl get svc
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
ingress-nginx   NodePort    10.110.153.160   <none>        80:30737/TCP,443:31369/TCP   21m
nginx-svc       ClusterIP   10.102.2.21      <none>        80/TCP                       28s
[root@k8s-master01 ~]# curl 10.102.2.21
Hello MyApp | Version: v1 | <a href="hostname.html">Pod Name</a>
[root@k8s-master01 ~]# vim ingress1.yaml
[root@k8s-master01 ~]# kubectl apply -f ingress1.yaml
ingress.extensions/nginx-test unchanged
[root@k8s-master01 ~]# kubectl get svc -n ingress-nginx
NAME                                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
default-http-backend                 ClusterIP   10.97.132.150   <none>        80/TCP
14h
ingress-nginx-controller             NodePort    10.97.38.110    <none>        80:32488/TCP,443:31176/TCP
14h
ingress-nginx-controller-admission   ClusterIP   10.97.202.183   <none>        443/TCP
14h
[root@k8s-master01 ~]# kubectl get svc
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
ingress-nginx   NodePort    10.110.153.160   <none>        80:30737/TCP,443:31369/TCP   32m
nginx-svc       ClusterIP   10.102.2.21      <none>        80/TCP                       11m
```

修改 windows



添加 192.168.66.10 www1.atguigu.com

在浏览器打开：域名：ip

[root@k8s-master01 ~]# **mkdir ingress-vh**
[root@k8s-master01 ~]# cd ingress-vh
[root@k8s-master01 ingress-vh]# **vim deployment1.yaml**
[root@k8s-master01 ingress-vh]# **kubectl get svc -n ingress-nginx**

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
default-http-backend	ClusterIP	10.97.132.150	<none>	80/TCP	14h
ingress-nginx-controller	NodePort	10.97.38.110	<none>	80:32488/TCP,443:31176/TCP	14h
ingress-nginx-controller-admission	ClusterIP	10.97.202.183	<none>	443/TCP	14h

[root@k8s-master01 ~]# **kubectl apply -f deployment1.yaml**
deployment.extensions/deployment1 created
service/svc-1 created
[root@k8s-master01 ~]# **kubectl get svc**

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
ingress-nginx	NodePort	10.110.153.160	<none>	80:30737/TCP,443:31369/TCP	57m
svc-1	ClusterIP	10.107.65.41	<none>	80/TCP	6s

[root@k8s-master01 ~]# **curl 10.107.65.41**
Hello MyApp | Version: v1 | Pod Name
[root@k8s-master01 ~]# **vim deployment2.yaml**
[root@k8s-master01 ~]# **kubectl apply -f deployment2.yaml**
deployment.extensions/deployment2 created
service/svc-2 created
[root@k8s-master01 ~]# **kubectl get svc**

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
ingress-nginx	NodePort	10.110.153.160	<none>	80:30737/TCP,443:31369/TCP	62m
svc-1	ClusterIP	10.107.65.41	<none>	80/TCP	4m20s
svc-2	ClusterIP	10.108.81.163	<none>	80/TCP	5s

[root@k8s-master01 ~]# **curl 10.108.81.163**
Hello MyApp | Version: v2 | Pod Name
[root@k8s-master01 ~]# **vim ingressrule.yaml**
[root@k8s-master01 ~]# **kubectl apply -f ingressrule.yaml**
ingress.extensions/ingresss1 created
ingress.extensions/ingresss2 created
修改 windows hosts 文件
192.168.66.10 www1.atguigu.com

192.168.66.10 www2.atguigu.com
浏览器分别找

[root@k8s-master01 ~]# **kubectl get pod**

NAME READY STATUS RESTARTS AGE

myapp-deploy-6cc7c66999-jnzj9 1/1 Running 1 20h

nginx-dm-7d967c7ff5-8cljq 1/1 Running 0 59m

nginx-dm-7d967c7ff5-mp7r6 1/1 Running 0 59m

[root@k8s-master01 ~]# **kubectl get pod -n ingress-nginx**

NAME READY STATUS RESTARTS AGE

default-http-backend-c858dffd-gvrnl 0/1 ImagePullBackOff 0 14h

ingress-nginx-admission-create-hj9qd 0/1 Completed 0 14h

ingress-nginx-admission-patch-gjjz6 0/1 Completed 0 14h

ingress-nginx-controller-5575c6cd9d-xrq4r 1/1 Running 0 14h

nginx-ingress-controller-7cbb74c44b-q6x26 1/1 Running 0 14h

[root@k8s-master01 ~]# **kubectl exec nginx-ingress-controller-7cbb74c44b-q6x26 -n ingress-nginx -it -- /bin/bash**

www-data@nginx-ingress-controller-7cbb74c44b-q6x26:/etc/nginx$

www-data@nginx-ingress-controller-7cbb74c44b-q6x26:/etc/nginx$

www-data@nginx-ingress-controller-7cbb74c44b-q6x26:/etc/nginx$ **ls**

fastcgi.conf fastcgi_params.default koi-win mime.types.default nginx.conf owasp-modsecurity-crs template win-utf

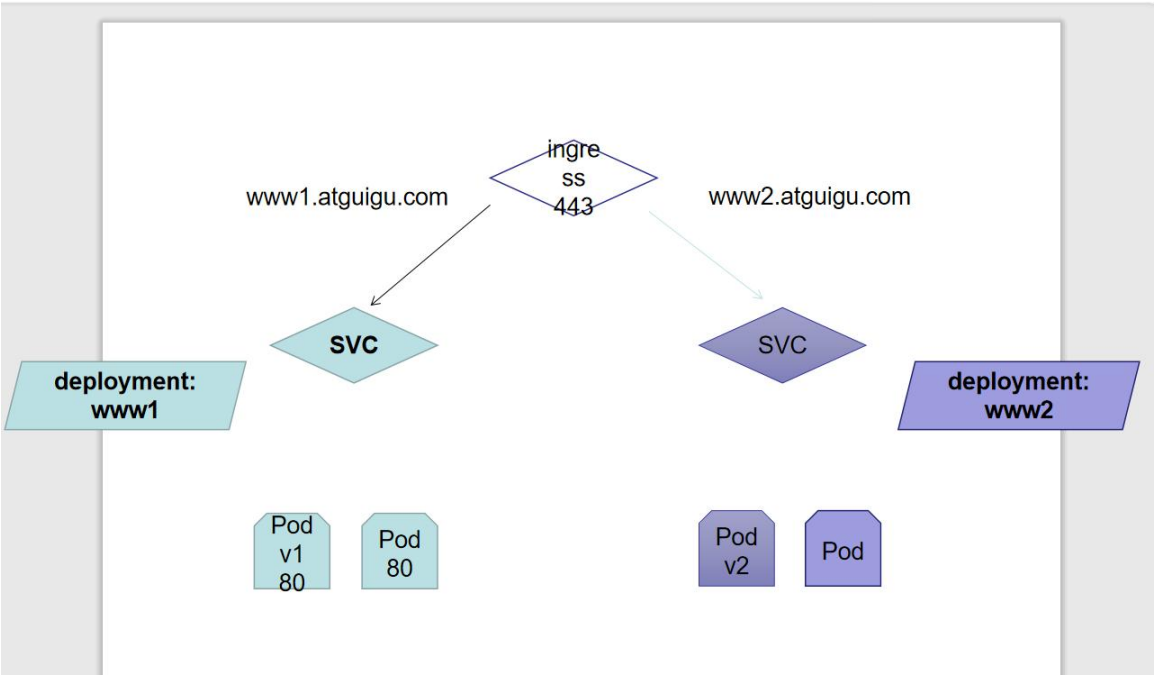
fastcgi.conf.default geoip lua modsecurity nginx.conf.default scgi_params uwsgi_params

fastcgi_params koi-utf mime.types modules opentracing.json scgi_params.default uwsgi_params.default

www-data@nginx-ingress-controller-7cbb74c44b-q6x26:/etc/nginx$ **cat nginx.conf**

```
    }
    ## end server www1.atguigu.com

    ## start server www2.atguigu.com
    server {
            server_name www2.atguigu.com ;
```



两个不同的基于域名的虚拟主机

HTTPS 代理

如果想要网站访问更安全，可以考虑带加密和认证的 https 协议。在客户端和 nginx 之间走 https 协议，nginx 反向代理到后端的时候因为是内网，还是走未加密的 http 协议即可。Https 的工作方式这里不额外介绍了，要完成配置，我们需要服务端的自签名 CA 证书和私钥文件。

[root@k8s-master01 https]# **openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -keyout tls.key -out tls.crt -subj "/CN=nginxsvc/O=nginxsvc"**（纸）

Generating a 2048 bit RSA private key

.....+++

.....+++

writing new private key to 'tls.key'

[root@k8s-master01 https]# **kubectl create secret tls tls-secret --key tls.key --cert tls.crt**

secret/tls-secret created

[root@k8s-master01 https]# **vim ../deployment1.yaml deployment3.yaml**

还有 2 个文件等待编辑(用之前的修改一下作为 deployment3）

[root@k8s-master01 https]# **kubectl apply -f deployment3.yaml**

deployment.extensions/deployment3 created

service/svc-3 created

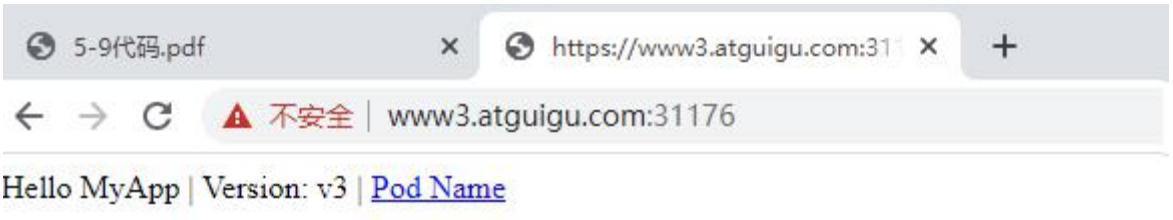
[root@k8s-master01 https]# **kubectl get svc**

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
ingress-nginx	NodePort	10.110.153.160	<none>	80:30737/TCP,443:31369/TCP	4h11m
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	2d22h
svc-1	ClusterIP	10.107.65.41	<none>	80/TCP	3h13m
svc-2	ClusterIP	10.108.81.163	<none>	80/TCP	3h9m
svc-3	**ClusterIP**	**10.103.188.60**	**<none>**	**80/TCP**	**10s**

[root@k8s-master01 https]# **curl 10.103.188.60**
Hello MyApp | Version: v3 | Pod Name
[root@k8s-master01 https]# **vim ingress-https.yaml**
[root@k8s-master01 https]# **kubectl apply -f ingress-https.yaml**（见纸）
ingress.extensions/https created
[root@k8s-master01 https]# **kubectl get svc -n ingress-nginx**

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
default-http-backend	ClusterIP	10.97.132.150	<none>	80/TCP	17h
ingress-nginx-controller	NodePort	10.97.38.110	<none>	80:32488/TCP,443:**31176**/TCP	17h
ingress-nginx-controller-admission	ClusterIP	10.97.202.183	<none>	443/TCP	17h

修改 window hosts
浏览器输入 https://www3.atguigu.com:31176

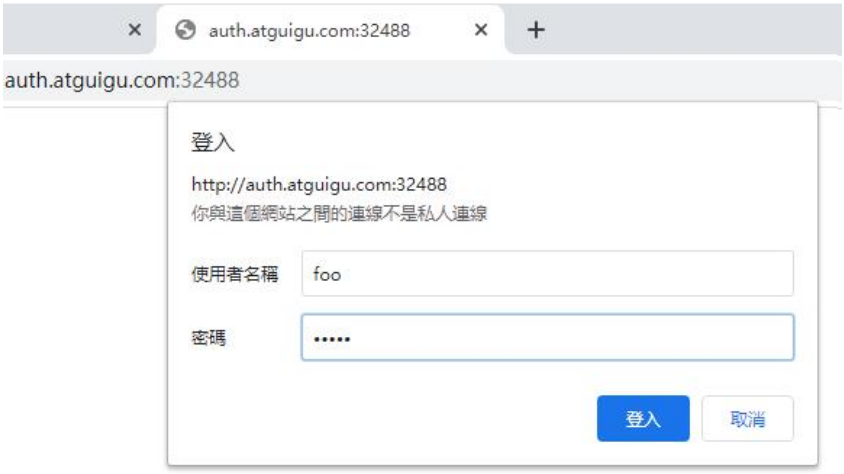


nginx 进行 basicauth 基础认证

（纸）
[root@k8s-master01 https]# **yum -y install httpd**
已加载插件：fastestmirror
[root@k8s-master01 https]# **cd**
[root@k8s-master01 ~]# **mkdir basic-auth**
[root@k8s-master01 ~]# **cd basic-auth/**
[root@k8s-master01 basic-auth]# **ls**
[root@k8s-master01 basic-auth]# **htpasswd -c auth foo**
New password:
Re-type new password:
Adding password for user foo
[root@k8s-master01 basic-auth]# **ls**
auth
[root@k8s-master01 basic-auth]# **kubectl create secret generic basic-auth --from-file=auth**
secret/basic-auth created
[root@k8s-master01 basic-auth]# **vim ingress.yaml**
[root@k8s-master01 basic-auth]# kubectl **apply -f ingress.yaml**
ingress.extensions/ingress-with-auth created
[root@k8s-master01 basic-auth]# **kubectl get svc -n ingress-nginx**

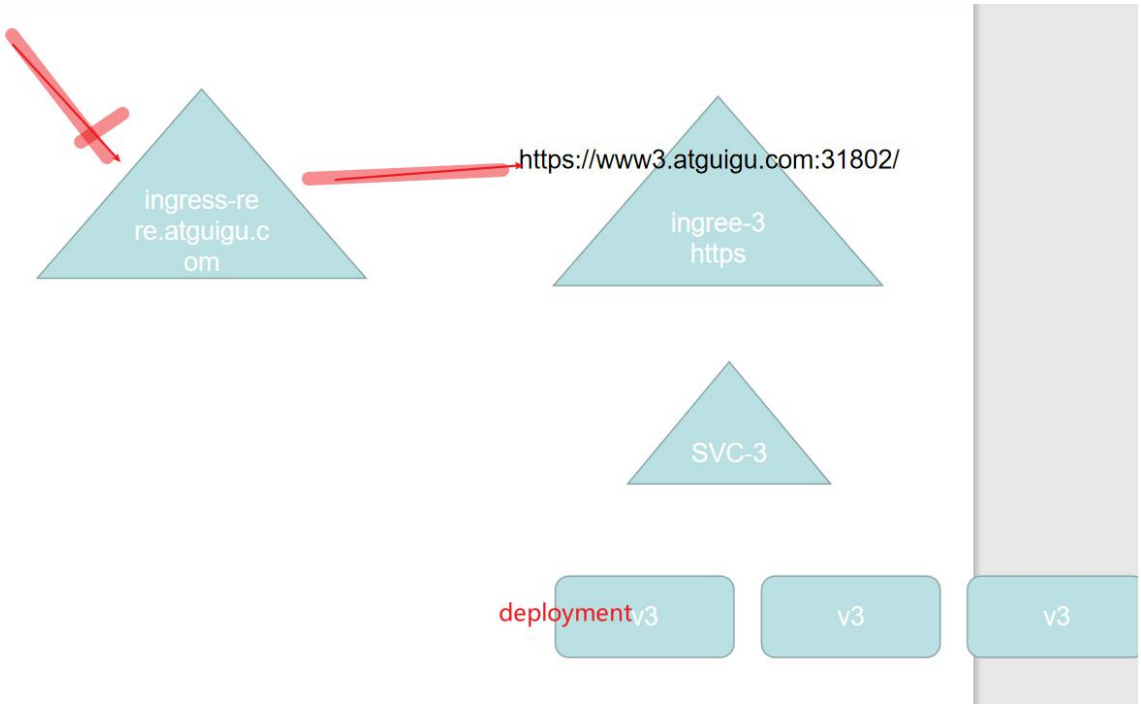
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
default-http-backend	ClusterIP	10.97.132.150	<none>	80/TCP	18h

ingress-nginx-controller NodePort 10.97.38.110 <none> **80:32488**/TCP,443:31176/TCP 18h
ingress-nginx-controller-admission ClusterIP 10.97.202.183 <none> 443/TCP 18h
修改 windows
192.168.66.10 auth.atguigu.com



Nginx 重写

上面 yaml 文件中的 annotations 除了可以声明认证方式外，还有很多重写功能。



访问 re.atguigu.com 可以跳转到 https://www3.atguigu.com:31176

[root@k8s-master01 ~]# **mkdir re**

[root@k8s-master01 ~]# **cd re**

[root@k8s-master01 re]# **vim re.yaml（纸）**

[root@k8s-master01 re]# **kubectl apply -f re.yaml**

ingress.extensions/nginx-test configured

[root@k8s-master01 re]# **kubectl get svc -n ingress-nginx**

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
default-http-backend	ClusterIP	10.97.132.150	<none>	80/TCP	18h
ingress-nginx-controller	NodePort	10.97.38.110	<none>	80:32488/TCP,443:31176/TCP	18h
ingress-nginx-controller-admission	ClusterIP	10.97.202.183	<none>	443/TCP	18h

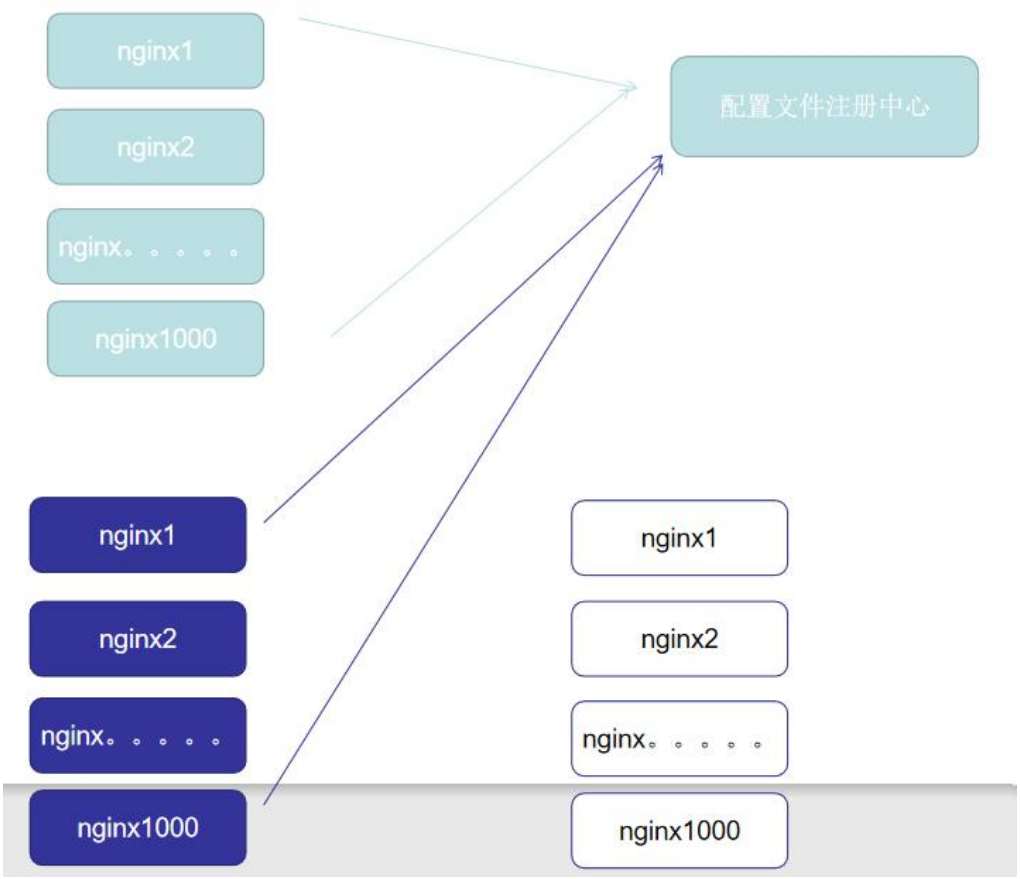
[root@k8s-master01 re]#

跳转

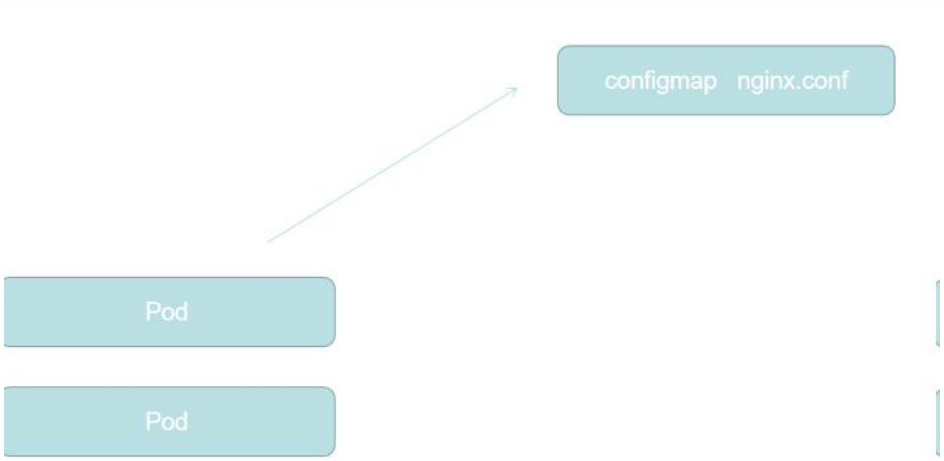
跳转的目的地址要写完整

同时在 hosts 文件中将 redirect.xiaofu.com 映射到任意 node 的 ip，此时访问 redirect.xiaofu.com:31958/会自动跳转到 mynginx.xiaofu.com:31958

文章目录



配置文件不一样，管理费劲，
服务项访问向配置文件中心索要，尽心分配
不同集群索要到不同的配置文件
更改是



申请同一个

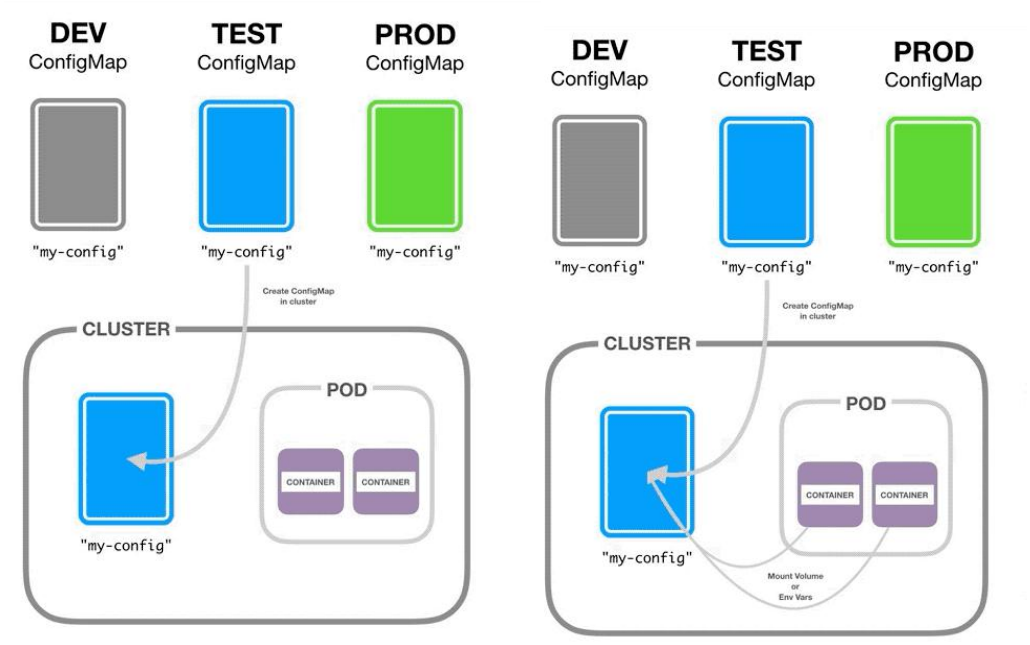
ConfigMap

三种创建方式

ConfigMap 是 k8s 集群中一个单独的资源，用来存储键值对形式的配置信息。通过将业务代码和配置信息隔离开来，可以减少容器的复杂度，同时让容器能动态部署到不同环境（开发，测试，生产），以及在不影响容器运行下进行配置的热更新。

但是因为 ConfigMap 中的数据都是用明文存储，不应该存放密码等私密信息。私密信息可以通过下一节要学习的 Secret 来进行存储。

下面这个动态图是一个很简单的 ConfigMap 工作原理的说明



首先，需要准备多个不同 ConfigMap，这里是一个环境一份

然后，根据需要，将合适环境的 ConfigMap 加入到 k8s 集群中
最后，pod 中的容器去 ConfigMap 中检索需要的内容
生成 ConfigMap
这里分两种情况来看。如果是类似 nginx.conf 这种大篇的配置信息文件，考虑用文件或者文件夹去生成 ConfigMap，方便后期维护，这时 ConfigMap 的 key 是文件名，value 是文件内容；如果只是三两个环境变量，考虑用 yaml 文件或者直接命令行生成 ConfigMap，简单快捷，这时 ConfigMap 是键值对和 yaml 文件或者命令行中的一致。

使用目录创建

（纸）

```
[root@k8s-master01 dr]#    kubectl create configmap game-config1 --from-file=../dr
configmap/game-config1 created
[root@k8s-master01 dr]# kubectl get configmaps game-config1 -o yaml
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: "2020-08-06T00:58:41Z"
  name: game-config1
  namespace: default
  resourceVersion: "294136"
  selfLink: /api/v1/namespaces/default/configmaps/game-config1
  uid: 9640b592-7be2-4535-8106-8bd3ac47e25f
```

使用文件创建

[root@k8s-master01 dr]# **kubectl delete configmaps game-config-2**
configmap "game-config-2" deleted
[root@k8s-master01 dr]# **kubectl create configmap game-config-2 --from-file=./game.properties**
configmap/game-config-2 created
[root@k8s-master01 dr]# **kubectl get configmaps game-config-2 -o yaml**

```
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
kind: ConfigMap
metadata:
  creationTimestamp: "2020-08-06T01:01:12Z"
  name: game-config-2
  namespace: default
  resourceVersion: "294388"
  selfLink: /api/v1/namespaces/default/configmaps/game-config-2
  uid: a854c33d-5ef3-4176-981a-8e29d71e33e7
```

使用字面创建

[root@k8s-master01 dr]# **kubectl create configmap special-config --from-literal=special.how=very --from-literal=special.type=charm**
configmap/special-config created
[root@k8s-master01 dr]# **kubectl describe configmap special-config**

```
Name:         special-config
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
special.how:
```

---
very
special.type:
---
charm
Events: <none>

在 pod 中使用 configmap

1.用 configmap 代替环境变量

[root@k8s-master01 dr]# **cd ..**
[root@k8s-master01 configmap]# **mkdir env**
[root@k8s-master01 configmap]# **cd env/**
[root@k8s-master01 env]# **vim env.yaml**
[root@k8s-master01 env]# **kubectl apply -f env.yaml**
configmap/env-config created
[root@k8s-master01 env]# **kubectl get cm**
NAME DATA AGE
env-config 1 8s
game-config 2 13m
game-config-2 1 9m48s
special-config 2 4m31s
[root@k8s-master01 env]# **kubectl describe cm env-config**
Name: env-config
Namespace: default
Labels: <none>
Annotations: kubectl.kubernetes.io/last-applied-configuration:

{"apiVersion":"v1","data":{"log_level":"INFO"},"kind":"ConfigMap","metadata":{"annotations":{},
"name":"env-config","namespace":"default"}}

Data
====
log_level:

INFO
Events: <none>
[root@k8s-master01 env]# **kubectl get cm special-config**
NAME DATA AGE
special-config 2 6m54s
[root@k8s-master01 env]# **kubectl describe cm special-config**
Name: special-config
Namespace: default
Labels: <none>

Annotations: <none>

Data
====
special.type:
----
charm
special.how:
----
very
Events: <none>
[root@k8s-master01 env]# **vim pod.yaml**
[root@k8s-master01 env]# **kubectl create -f pod.yaml**
pod/dapi-test-pod created
[root@k8s-master01 env]# **kubectl get pod**
NAME READY STATUS RESTARTS AGE
dapi-test-pod 0/1 Completed 0 5s
[root@k8s-master01 env]# **kubectl log dapi-test-pod**
log is DEPRECATED and will be removed in a future version. Use logs instead.
MYSERVICE_SERVICE_HOST=10.98.77.160
MYAPP_SERVICE_PORT_HTTP=80
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_SERVICE_PORT=443
MYDB_SERVICE_PORT=80
MYDB_PORT=tcp://10.107.52.95:80
INGRESS_NGINX_PORT_80_TCP_ADDR=10.110.153.160
HOSTNAME=dapi-test-pod
...
log_level=INFO
...
[root@k8s-master01 env]# **kubectl log dapi-test-pod |grep TYPE**
log is DEPRECATED and will be removed in a future version. Use logs instead.
SPECIAL_TYPE_KEY=charm

2. 用 cm 设置命令行参数

[root@k8s-master01 env]# **vim pod1.yaml**
[root@k8s-master01 env]# **kubectl create -f pod1.yaml**
pod/dapi-test-pod66 created
[root@k8s-master01 env]# **kubectl get pod**
NAME READY STATUS RESTARTS AGE
dapi-test-pod 0/1 Completed 0 9m1s
dapi-test-pod66 0/1 Completed 0 75s
[root@k8s-master01 env]# **kubectl log dapi-test-pod66**

log is DEPRECATED and will be removed in a future version. Use logs instead.
very charm

3. 将文件填入数据卷

[root@k8s-master01 env]# **vim 111.yaml**
[root@k8s-master01 env]# **kubectl delete pod dapi-test-pod77**
pod "dapi-test-pod77" deleted
[root@k8s-master01 env]# **kubectl create -f 111.yaml**
pod/dapi-test-pod77 created
[root@k8s-master01 env]# **kubectl get pod**

NAME	READY	STATUS	RESTARTS	AGE
dapi-test-pod	0/1	Completed	0	102m
dapi-test-pod66	0/1	Completed	0	94m
dapi-test-pod77	0/1	Completed	0	7s

[root@k8s-master01 env]# **kubectl log dapi-test-pod7**7
log is DEPRECATED and will be removed in a future version. Use logs instead.
very[root@k8s-master01 env]# vim 111.yaml
[root@k8s-master01 env]# **kubectl exec dapi-test-pod77 -it -- /bin/sh**
/ #
/ #
/ # **cd /etc/config**
/etc/config # **ls**
special.how special.type
/etc/config # cat special.how
very/etc/config # cat special.type
charm/etc/config #
/etc/config # **exit**
command terminated with exit code 127

Configmap 的热更新

[root@k8s-master01 configmap]# **mkdir config**
[root@k8s-master01 configmap]# **cd config/**
[root@k8s-master01 config]# **ls**
[root@k8s-master01 config]# **vim 111.yaml**
[root@k8s-master01 config]# **kubectl apply -f 111.yaml**
configmap/log-config created
deployment.extensions/my-nginx created
[root@k8s-master01 config]# **kubectl get pod**

NAME	READY	STATUS	RESTARTS	AGE
my-nginx-856cb6c947-tlw25	1/1	Running	0	7s

[root@k8s-master01 config]# **kubectl exec my-nginx-856cb6c947-tlw25 -it -- cat /etc/config/log_level**
INFO[root@k8s-master01 config]# **kubectl edit configmap log-config**
configmap/log-config edited

[root@k8s-master01 config]# **kubectl exec my-nginx-856cb6c947-tlw25 -it -- cat /etc/config/log_level**

DEBUG

[root@k8s-master01 config]# **kubectl get pod**

NAME READY STATUS RESTARTS AGE

my-nginx-856cb6c947-tlw25 1/1 Running 0 8m9s

[root@k8s-master01 config]# **kubectl exec my-nginx-856cb6c947-tlw25 -it -- /bin/sh**

/ # **cd /run/secrets/kubernetes.io/serviceaccount**

/run/secrets/kubernetes.io/serviceaccount # **cd serviceaccount/**

/bin/sh: cd: can't cd to serviceaccount/: No such file or directory

/run/secrets/kubernetes.io/serviceaccount # **ls**

ca.crt namespace token

/run/secrets/kubernetes.io/serviceaccount # **cat ca.crt**

-----BEGIN CERTIFICATE-----

MIICyDCCAbCgAwIBAgIBADANBgkqhkiG9w0BAQsFADAVMRMwEQYDVQQDEwprdWJl

...

nE3bimJlBatrjbG915/RqlYhpFyHBMQBXic4CwD29Hggmp8QLlATLAFaosxDdjR9

-----END CERTIFICATE-----

/run/secrets/kubernetes.io/serviceaccount # **cat namespace**

default/run/secrets/kubernetes.io/serviceaccount # **cat token**

eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZ

...

6WRS0hJtGsx_Hl8iqFLjrJVQdxmC_k89TDD5f6yIJfFz-IZQkaZXjqd7ihNb0W6Gq4B72B6AqGhBmfLx1

gRKov8uRLyQyYRTJXHtpS1qEHG7eg/run/secrets/kubernetes.io/serviceaccount # exit

Secret

[root@k8s-master01 dr]# **kubectl create secret generic secret-1 --from-file=../dr**

secret/secret-1 created（可以用目录或文件创建）

[root@k8s-master01 dr]# **kubectl describe secret secret-1**

Name: secret-1

Namespace: default

Labels: <none>

Annotations: <none>

Type: Opaque

Data

====

game.properties: 158 bytes

ui.properties: 84 bytes（可以发现是查看不了里面内容的）

[root@k8s-master01 dr]# **kubectl get secret secret-1 -o yaml**

apiVersion: v1

data:

```
  game.properties:
ZW5lbWllcz1hbGllbnMKbGl2ZXM9MwplbmVtaWVzLmNoZWF0PXRydWUKZW5lbWllcy5jaGVhdC5sZXZlbD1ub0dvb2RSb3R0ZW4Kc2VjcmV0LmNvZGUucGFzc3BocmFzZT1VVURETFJMUkJBQkFTCnNlY3JldC5jb2RlLmFsbG93ZWQ9dHJ1ZQpzZWNyZXQuY29kZS5saXZlcz0zMAo=
  ui.properties:
Y29sb3IuZ29vZD1wdXJwbGUKY29sb3IuYmFkPXllbGxvdwphbGxvdy50ZXh0bW9kZT10cnVlCmhvdy5uaWNlLnRvLmxvb2s9ZmFpcmx5TmljZQoK
kind: Secret
metadata:
  creationTimestamp: "2020-08-06T01:33:57Z"
  name: secret-1
  namespace: default
  resourceVersion: "297705"
  selfLink: /api/v1/namespaces/default/secrets/secret-1
  uid: 454e1e9c-7d6b-49b2-9d0f-1e0beccd2f04
type: Opaque
```

Opaque Secret

```
[root@k8s-master01 config]# echo -n "admin" | base64
YWRtaW4=
[root@k8s-master01 config]# echo -n "1f2d1e2e67df" | base64
MWYyZDFlMmU2N2Rm
[root@k8s-master01 config]# vim secret.yaml
[root@k8s-master01 config]# kubectl apply -f secret.yaml
secret/mysecret created
[root@k8s-master01 config]# kubectl get secret
NAME                  TYPE                                  DATA   AGE
basic-auth            Opaque                                1      5h23m
default-token-djvlb   kubernetes.io/service-account-token   3      3d4h
mysecret              Opaque                                2      10s
tls-secret            kubernetes.io/tls                     2      6h6m
```

和 ConfigMap 一样，也是可以通过环境变量和 volume 的方式去使用 Secret。环境变量适合命令行需要输入密码等场景，而 volume 适合用文件去进行远端验证场景。

把 Secret 挂载到 volume 中

```
[root@k8s-master01 config]# vim pod1.yaml
[root@k8s-master01 config]# kubectl apply -f pod1.yaml
pod/seret-test created
[root@k8s-master01 config]# kubectl get pod
NAME                        READY   STATUS    RESTARTS   AGE
my-nginx-856cb6c947-tlw25   1/1     Running   0          21m
seret-test                  1/1     Running   0          6s
```

[root@k8s-master01 config]# **kubectl exec seret-test -it -- /bin/sh**
/ # **cd /etc/secrets**
/etc/secrets # **ls**
password username
/etc/secrets # **exit**

将 secret 导到环境变量

[root@k8s-master01 config]# **vim env.yaml**
[root@k8s-master01 config]# **kubectl apply -f env.yaml**
deployment.extensions/pod-deployment created
[root@k8s-master01 config]# **kubectl get pod**

NAME	READY	STATUS	RESTARTS	AGE
pod-deployment22-7958ddcc9d-bbjrg	**1/1**	**Running**	**0**	**4s**
pod-deployment22-7958ddcc9d-gw9wg	1/1	Running	0	4s
seret-test	1/1	Running	0	7m3s

[root@k8s-master01 config]# **kubectl exec pod-deployment22-7958ddcc9d-nz7v7 -it -- /bin/sh**
/ # echo $TEST_PASSWORD
1f2d1e2e67df

/ # e**xit**

私有仓库认证下载

[root@k8s-master01 config]# **docker rmi hub.atguigu.com/library/mynew:v1**
Error: No such image: hub.atguigu.com/library/mynew:v1
[root@k8s-master01 config]# **docker tag perl hub.atguigu.com/mimi/newperl:v5**
[root@k8s-master01 config]# **docker push hub.atguigu.com/mimi/newperl:v5**
[root@k8s-master01 config]# **docker images**

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
quay.io/coreos/flannel	v0.12.0-amd64	4e9f801d2217	4 months ago	52.8MB
registry.aliyuncs.com/google_containers/nginx-ingress-controller	0.26.1	29024c9c6e70	10 months ago	483MB
perl	latest	ac0fb8cfc61a	11 months ago	858MB
hub.atguigu.com/mimi/newperl	v5	ac0fb8cfc61a	11 months ago	858MB
k8s.gcr.io/kube-proxy	v1.15.1	89a062da739d	12 months ago	82.4MB
k8s.gcr.io/kube-scheduler	v1.15.1	b0b3c4c404da	12 months ago	81.1MB
k8s.gcr.io/kube-apiserver	v1.15.1	68c3eb07bfc3	12 months ago	207MB

k8s.gcr.io/kube-controller-manager v1.15.1
d75082f1d121 12 months ago 159MB
k8s.gcr.io/coredns 1.3.1
eb516548c180 18 months ago 40.3MB
k8s.gcr.io/etcd 3.3.10
2c4adeb21b4f 20 months ago 258MB
k8s.gcr.io/pause 3.1
da86e6ba6ca1 2 years ago 742kB
[root@k8s-master01 config]# **docker rmi -f ac0fb8cfc61a**
Untagged: perl:latest
Untagged: hub.atguigu.com/mimi/newperl:v5
Untagged:
hub.atguigu.com/mimi/newperl@sha256:a1c89e0095f6972492be31892141ef24c795a9a8c99c7
24e7f869f8e4b1565b9
Deleted: sha256:ac0fb8cfc61a3e7767284a82c840b47f66f14bfbcd4c9d361e50a2ad0932323d
Deleted: sha256:94498452f378e97c52bc2dc1ea05bac3af97da117056d69e833e26603cf5ce09
Deleted: sha256:c73dcd2001412ba861e942d43580dc79b1fe3e5cd14127b4f60a595d7303fc25
Deleted: sha256:81cfb26f96d99dbf2f9610c90290cd614ec0e92c7a5a17f4fd01d5007e758074
Deleted: sha256:86c2a34929fb12d692141f9004935c02ef70450cf8eb3e04c364bb4260ce7c1b
Deleted: sha256:fe5941d414f1fd07ffbbbc1af94133868316b9b4c14d65cc6ce541898ed375d8
Deleted: sha256:6d28eb79bc667b61ccdb4ec83bb27da3f05eed64dabbccb5d97426264b524de6
Deleted: sha256:660314270d7683a945d0cc83e5561d61f022e7e6731fd2e3fad04a6544bf42d8
查找设成私密为什么还能 **pull**
[root@k8s-master01 config]# **cd /etc/docker**
[root@k8s-master01 docker]# **ls**
daemon.json key.json
[root@k8s-master01 docker]# **cat key.json** 记住密钥？
{"crv":"P-256","d":"2YRD_WtvFfqfgdZsRQlto09NVOdstFnbru-Fos_dE2U","kid":"6GHX:PGPX:BY3N:
7YO3:WQFR:L25H:GVE5:LM3J:YDU2:IVQZ:RCME:PL2K","kty":"EC","x":"_EC6KIa5U6gd7AjLliFbfT-l
YBQypSX9IZgH0P4i5yQ","y":"fewepmknBjFRqEfQmd-W3NWj4kx5x0-tawE0-6ehDRE"}[root@k8s-
master01 docker]# **docker images** 镜像有？
REPOSITORY TAG
IMAGE ID CREATED SIZE
quay.io/coreos/flannel v0.12.0-amd64
4e9f801d2217 4 months ago 52.8MB
registry.aliyuncs.com/google_containers/nginx-ingress-controller 0.26.1
29024c9c6e70 10 months ago 483MB
k8s.gcr.io/kube-proxy v1.15.1
89a062da739d 12 months ago 82.4MB
k8s.gcr.io/kube-scheduler v1.15.1
[root@k8s-master01 docker]# **docker logout hub.atguigu.com** 登出，每个节点都需要
Removing login credentials for hub.atguigu.com
[root@k8s-master01 docker]# **docker pull hub.atguigu.com/mimi/newperl:v5**
Error response from daemon: pull access denied for hub.atguigu.com/mimi/newperl, repository

does not exist or may require 'docker login': denied: requested access to the resource is denied

```
[root@k8s-master01 docker]# cd
[root@k8s-master01 ~]# mkdir reg
[root@k8s-master01 ~]# cd reg/
[root@k8s-master01 reg]# vim pod.yaml
[root@k8s-master01 reg]# kubectl create -f pod.yaml
pod/foo created
[root@k8s-master01 reg]# kubectl get pod
NAME                                    READY   STATUS             RESTARTS   AGE
foo                                     0/1     ImagePullBackOff   0          6s
[root@k8s-master01 reg]# kubectl describe pod foo
Name:               foo
Namespace:          default
Priority:           0
。...
  Warning  Failed     20s (x3 over 65s)  kubelet, k8s-node02  Error: ErrImagePull
  Normal   BackOff    9s (x4 over 64s)   kubelet, k8s-node02  Back-off pulling image
"hub.atguigu.com/mimi/newperl:v5"
  Warning  Failed     9s (x4 over 64s)   kubelet, k8s-node02  Error: ImagePullBackOff
[root@k8s-master01 reg]# kubectl create secret docker-registry myregistrykey
--docker-server=hub.atguigu.com --docker-username=admin
--docker-password=Harbor12345 --docker-email=wangyanglinux@163.com
secret/myregistrykey created
[root@k8s-master01 reg]# vim pod.yaml
[root@k8s-master01 reg]# kubectl delete pod foo
pod "foo" deleted
[root@k8s-master01 reg]# kubectl create -f pod.yaml
pod/foo created
[root@k8s-master01 reg]# kubectl get pod
NAME                                    READY   STATUS             RESTARTS   AGE
foo                                     0/1     CrashLoopBackOff   1          5s
```

注意这里的 echo 和后面的打印内容不要分开，不要写成[“echo”,“xxx”]

但是值得说明的是，虽然配置文件可以热更新，并不代表容器内的应用也支持热更新。例如 nginx 的配置文件修改了，还是需要重启一下 nginx 才会去重新读取配置信息。这就是具体应用的问题了，和我们这里的 ConfigMap 无关。修改 ConfigMap，环境变量不会热更新，只有 volume 对应的文件会热更新
文件热更新不代表容器内应用会去实时读取新的文件，要具体应用具体分
同时还要注意，修改了 ConfigMap，其对应的容器内环境变量并不会热更新。

secret 的潜在问题

虽然说 k8s 提供了 secret 这种方式去存储私密信息，但是其实还是有很多方面做的并不好

etcd 安全性 - secret 都以未加密形式保存在 etcd 中，需要用户去额外考虑 etcd 的加密和访问权限等问题
容器安全性 - 不能有效确保容器在使用 secret 的过程中不会泄露出去
加密性 - k8s 里面的 secret 都是用 base64 编码保存，极易可逆还原，不够安全
鉴于这些原因，越来越多的 k8s 集群开始引入第三方的 secret 管理方案，其中 Hashcorp 公司的 Vault 是比较流行的开源解决方案。

Hashicorp Vault provides secrets management and data protection, with advanced features like dynamic secrets, namespaces, leases, and revocation for secrets data

Docker 中的数据保存

docker 启动容器，只要容器不被删除其内部的文件不会消失

[root@k8s-node1 ~]# **docker run -it -d victor2019/test:v1**
e64c2bc31ac04174c1c78f9223e01a2cdec371104c58b74bdfc0a439913bf2d3
之后往容器内写点东西，因为我这个测试镜像是 nginx，所以写到主页面便于测试

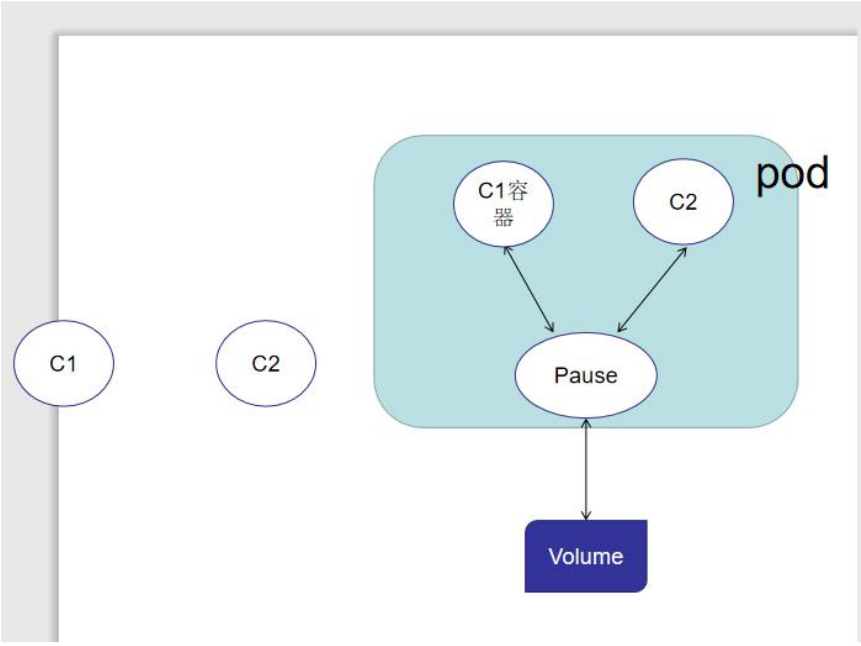
[root@k8s-node1 ~]# **docker exec e64c bash -c "echo hello > /usr/share/nginx/html/index.html"**
查看一下容器的 docker0 网段的 ip

[root@k8s-node1 ~]# docker inspect e64c | grep 172
"Gateway": "172.17.0.1",
"IPAddress": "172.17.0.2",
"Gateway": "172.17.0.1",
"IPAddress": "172.17.0.2",
验证下写进去的内容

[root@k8s-node1 ~]# **curl 172.17.0.2**
hello
然后停掉容器再启动
[root@k8s-node1 ~]# **docker kill e64c**
e64c
[root@k8s-node1 ~]# **docker start e64c**
e64c
[root@k8s-node1 ~]# **curl 172.17.0.2**

hello
容器内的文件一直都还在。
所以直接用 docker 起容器的话，指定--restart=always 就可以保证容器提供持续服务了。

K8s 中的数据保存



C1,c2 共享存储卷
volume 就是一个可以被 pod 中所有容器使用的公共文件夹，每个容器可以将这个公共文件夹挂载到本容器内的一个目录（不同容器的目录可以不同），这样 pod 内容器就可以互相交换数据。volume 的生命周期和 pod 一致，所以不管 pod 重启多少次 volume 内的内容和挂载目录都不会改变，但是如果 pod 被终止 volume 就会跟着消失。

注意这里的重启是指因为容器崩溃导致的重启，而不是手动删除 pod 然后因为重启策略而进行的重启
注意这里的重启是指因为容器崩溃导致的重启，而不是手动删除 pod 然后因为重启策略而进行的重启

Emptydir

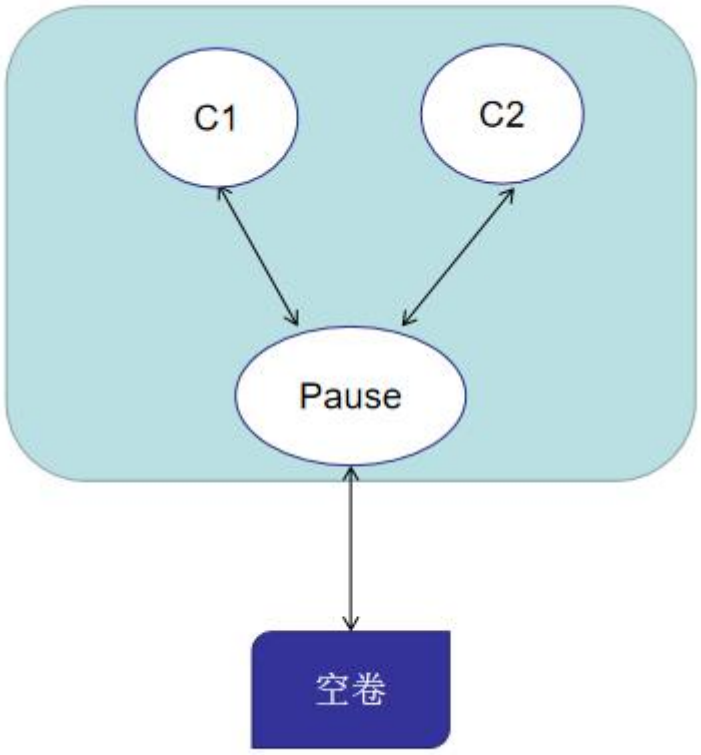
当一个 pod 被分配到 node 时，一个 emptyDir volume 首先被创建，并且其生命周期和 pod 一样长。正如其名字表示的那样，emptyDir 一开始是空的，pod 中的所有容器都可以在里面进行读写。如果 pod 在 node 上被删除，emptyDir 中的数据也会被永久删除。

容器崩溃不会将 pod 从 node 上删除，而只是重启，所以 emptyDir 对容器崩溃来说是安全的

emptyDir 通常用于声明一些临时空间，存放一些临时文件，尤其是多容器配合操作的时候。或者是容器从崩溃恢复时候的一些 checkpoint。

两个配置字段在 pod.spec.volumes.emptyDir 中声明

字段	类型	说明
medium	string	默认是空字符串表示硬盘，还可以是 Memory 表示内存
sizeLimit	string	默认是 nil 表示不指定大小



[root@k8s-master01 ~]# **mkdir volumn**
[root@k8s-master01 ~]# **cd volumn**
[root@k8s-master01 volumn]# **vim em.yaml（纸）**
[root@k8s-master01 volumn]# **kubectl get pod**

NAME	READY	STATUS	RESTARTS	AGE
test-pd	**1/1**	**Running**	**0**	**6s**

[root@k8s-master01 volumn]# **kubectl exec test-pd -it -- /bin/sh**
/ # cd /cache/
/cache # **ls**
/cache # exit
[root@k8s-master01 volumn]# **ls**
em.yaml
[root@k8s-master01 volumn]# **vim em.yaml（创建两个 container）**
[root@k8s-master01 volumn]# **kubectl apply -f em.yaml**

pod/test-pd1 created

[root@k8s-master01 volumn]# **kubectl get pod**

NAME READY STATUS RESTARTS AGE

test-pd1 2/2 Running 0 36s

[root@k8s-master01 volumn]# **kubectl exec test-pd1 -c test-container -it -- /bin/sh**

/ # ls

bin cache dev etc home lib media mnt opt proc root run sbin srv sys tmp usr var

/ # cd /cache

/cache # ls

/cache # **date>index.html**

/cache # **cat index.html**

Thu Aug 6 03:02:53 UTC 2020

/cache # **cat index.html**

Thu Aug 6 03:02:53 UTC 2020

Thu Aug 6 03:05:19 UTC 2020

/cache #

[root@k8s-master01 volumn]# **kubectl exec test-pd1 -c test-2 -it -- /bin/sh**

/ # ls

bin dev etc home proc root sys test tmp usr var

/ # cd /test

/test # **cat index.html**

Thu Aug 6 03:02:53 UTC 2020

/test # **date>>index.html**（此处追加，两边都可以看到）

/test # **cat index.html**

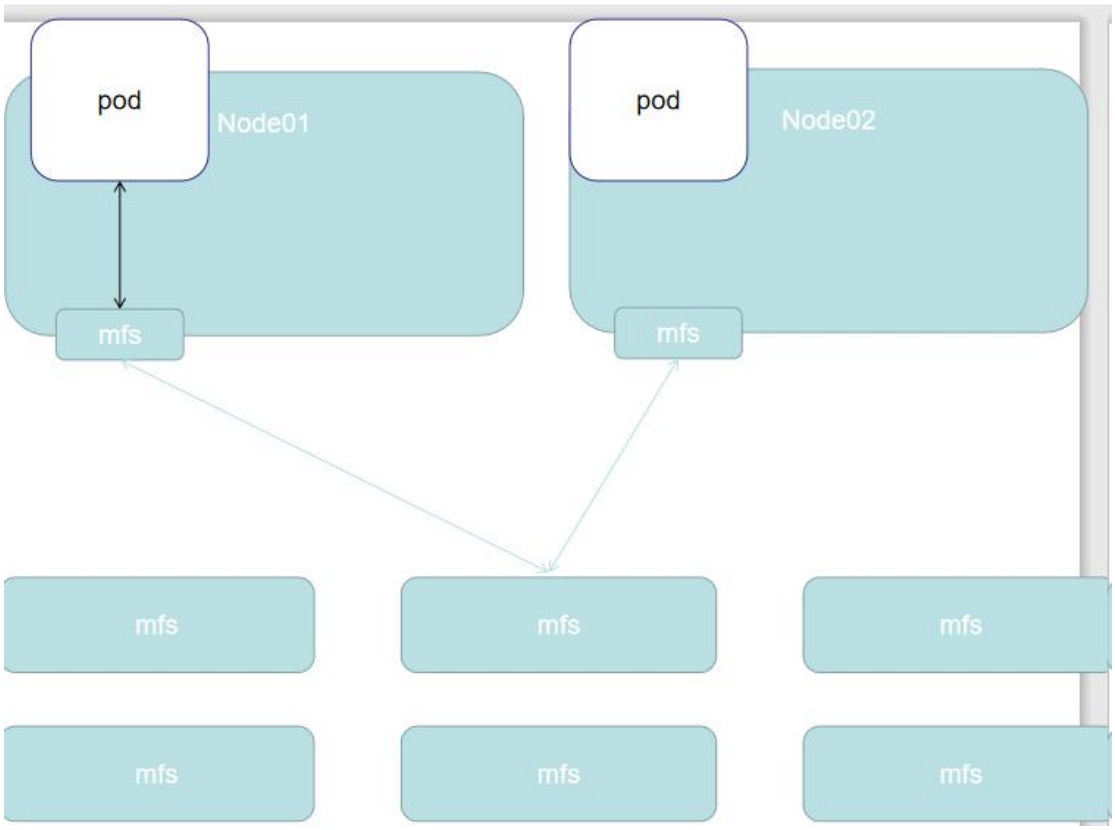
Thu Aug 6 03:02:53 UTC 2020

Thu Aug 6 03:05:19 UTC 2020

/test #

其中 **emptyDir: {}**表示用默认配置，也就是不声明大小的硬盘空间。

创建成功以后进入 **myalpine** 容器，往 **emptyDir** 绑定的目录写入文件，之后再访问另一个 **mynginx** 容器时被显示，说明因为 **emptyDir** 的引入一个 **pod** 内的容器可以共享同一个存储卷



Hostpath

hostPath 是将 node 中的文件或目录挂载成为 pod 的 volume。

这里看起来和 docker 里面的 volume 很像，但是要注意这里 node 上的内容并不会持久保存，要注意区分

除了 node 本地的文件系统，还可以使用 NFS 或者 MFS 这种网络存储。只要是能被 node 锚定的存储设备都可以被 hostPath 绑定，所以 hostPath 的使用范围是非常广泛的。

两个配置字段在 pod.spec.volumes.hostPath 中声明

字段　　类型　　说明
path string　　必填，node 上的目录
type string　　见下面单独表格

[root@k8s-master01 ~]# **cd volumn**
[root@k8s-master01 volumn]# **vim pod1.yaml**
[root@k8s-master01 volumn]# **kubectl apply -f pod1.yaml**
pod/test-pd created
[root@k8s-master01 volumn]# kubectl get pod
NAME　　READY　　STATUS　　RESTARTS　　AGE

test-pd1 **1/1** Running 0 27s

[root@k8s-master01 volumn]# **kubectl exec -it test-pd1 -it -- /bin/sh（因为只有一个 container 就没指定）**

/ # cd /test-pd

/test-pd # ls

/test-pd # **date>index.html**

/test-pd # **cat index.html**

Thu Aug 6 03:44:18 UTC 2020

2020 年 08 月 06 日 星期四 11:45:14 CST

/test-pd #

Node01 没有

[root@k8s-node01 data]# **cd /data/**

[root@k8s-node01 data]# **ls**

Node02 才有

[root@k8s-node02 ~]# **mkdir /data**

[root@k8s-node02 ~]# **cd /data**

[root@k8s-node02 data]# **ls**

index.html

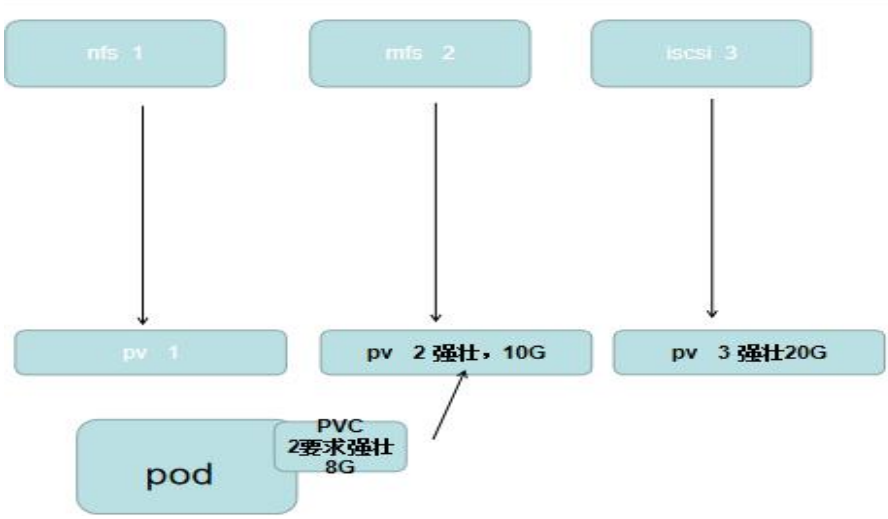
[root@k8s-node02 data]# **cat index.html**

Thu Aug 6 03:44:18 UTC 2020

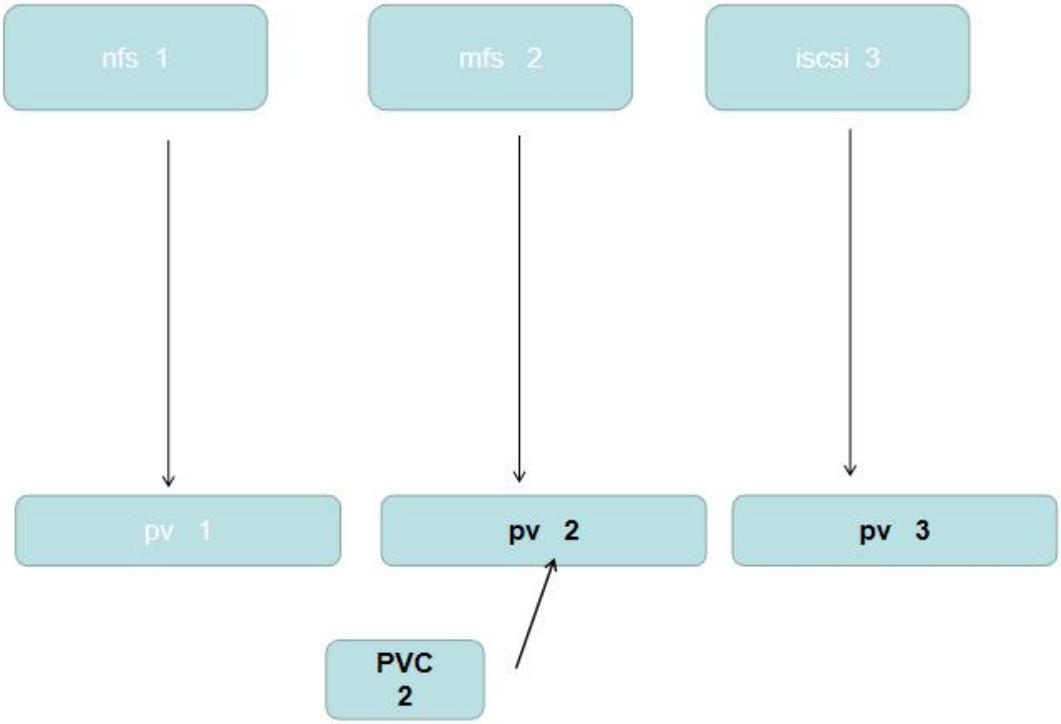
[root@k8s-node02 data]# **date>>index.html**

PVC

有存储工程师创建
应用工程师调用
Pv 速度大小是否满足要求
所以有 pvc 寻找满足的进行绑定



选择最小的与之匹配



请求绑定的类
后端存储有很多方案，慢，快，更快。分级，一类存储，二类存储。。。
可以指定 Pvc 请求的类

操作

在 harbor 安装 nfs
在各个节点安装 nfs
[root@k8s-master01 volumn]# **cd**
[root@k8s-master01 ~]# **mkdir /test**
[root@k8s-master01 ~]# **showmount -e 192.168.66.100**
Export list for 192.168.66.100:
/nfs *
[root@k8s-master01 ~]# **mount -t nfs 192.168.66.100:/nfs /test/**
[root@k8s-master01 ~]# **cd /test/**
[root@k8s-master01 test]# **ls**
[root@k8s-master01 test]#**vim 1.yaml**
[root@k8s-master01 test]# **cd**
[root@k8s-master01 ~]# **umount /test/**
[root@k8s-master01 ~]# **rm -rf /test/**
[root@k8s-master01 ~]# **ls**
anaconda-ks.cfg deployment2.yaml ingress-http.yaml
kubeadm-basic.images.tar.gz mydb.yaml pod.yaml svc-none.yaml

basic-auth deployment.yaml ingressrule.yaml kubernetes.conf myservice.yaml re svc.yaml
configmap external.yaml ingress-vh live-exec.yaml nodeport.yaml readiness.yaml volumn
cronbjob.yaml https ini-pod.yaml live.yaml perl.tar reg
daemonset.yaml ingree.contro.tar.zip.001 job.yaml load-images.sh perl.tar.gz rs.yaml
deployment1.yaml ingress1.yaml kubeadm-basic.images mkdir perl.tar.zip.001 svc-deployment.yaml

在 **harbor** 创建多个 **nfs**

[root@k8s-harbor ~]# **vim /etc/exports**
[root@k8s-harbor /]# **mkdir /nfs{1..3}**
[root@k8s-harbor /]# **ls**
bin boot data dev etc home lib lib64 media mnt nfs nfs1 nfs2 nfs3 nfsdata opt proc root run sbin srv sys tmp usr var
[root@k8s-harbor /]# **chmod 777 nfs1/ nfs2/ nfs3/**
[root@k8s-harbor /]# **chown nfsnobody nfs1/ nfs2/ nfs3/**
[root@k8s-harbor /]# **systemctl restart rpcbind**
[root@k8s-harbor /]# **systemctl restart nfs**
在节点看是否成功
[root@k8s-master01 pv]# **mkdir /test**
[root@k8s-master01 pv]# **mount -t nfs 192.168.66.100:/nfs1 /test**
[root@k8s-master01 pv]# **vim /test/index.html**
[root@k8s-master01 pv]# **umount /test/**
[root@k8s-master01 pv]# **rm -rf /test/**
[root@k8s-master01 pv]# **ls**
pv.yaml

创建多个 **pv**

[root@k8s-master01 pv]# **vim pv.yaml pv2.yaml**
还有 2 个文件等待编辑
[root@k8s-master01 pv]# **kubectl create -f pv2.yaml**
persistentvolume/nfspv2 created
persistentvolume/nfspv3 created
persistentvolume/nfspv4 created
persistentvolume/nfspv5 created
[root@k8s-master01 pv]# **kubectl get pv**
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE
nfspv1 10Gi RWO Retain Available nfs 19m

nfspv2	5Gi	ROX	Retain	Available	nfs
2m8s					
nfspv3	5Gi	RWX	Retain	Available	nfs
2m8s					
nfspv4	1Gi	ROX	Retain	Available	nfs
2m8s					
nfspv5	10Gi	RWO	Retain	Available	**slow**
3s					

总结

4 种创建 ConfigMap 的方式：文件/文件夹/yaml/命令行
3 种典型的 ConfigMap 使用场景：环境变量/命令参数/volume 热更新
需要注意
总结下 secret 的知识点

4 种创建 secret 的方法：文件/文件夹/yaml/命令行
2 种使用 secret 的方法：环境变量/volume
还有一种特殊的 secret 专门用于私有仓库的镜像拉取，可以通过认证文件或者命令行来创建
命令行的方式都不推荐，因为可以在 history 中查看
k8s 自带的 secret 还有很多不足，可以考虑第三方的 secret 管理方案，例如 Hashcorp Vault
总结
总结下 volume 的一些知识点

和 docker 的 volume 有本质区别，不能混淆
volume 的周期和 pod 一样，容器的崩溃导致的 pod 重启不会影响 volume 的内容
pod 的删除会将 volume 的内容永久删除
emptyDir 用来存储一些临时内容，和 node 上的存储无关
hostPath 用来将 node 的文件系统做为 volume，每个可能被分配的 node 都要准备好条件，尤其要注意 kubelet 的运行权限
volume 虽然给 pod 内容器的文件共享带来很多便利，但是这只是解决了一个痛点。如果我想 pod 即使被删除其容器的内容也依然会被持久化，又该如何做呢？下一节要学习的 Persistent Volume 来对 volume 进行一次升级。

节点亲和性

默认情况下 pod 被分配到哪个 node 都是随机的，但是很多情况下这不太符合预期。例如有多台 node，有的属于 cpu 密集型适合逻辑运算，有的属于 gpu 密集型适合机器学习。这时候就需要对 pod 调度的 node 有所规划，

软策略硬策略

[root@k8s-master01 pv]# **mkdir affi**

[root@k8s-master01 pv]# **rm -rf affi**

[root@k8s-master01 pv]# **cd**

[root@k8s-master01 ~]# **mkdir affi**

[root@k8s-master01 ~]# **cd affi**

[root@k8s-master01 affi]# **ls**

[root@k8s-master01 affi]# **vim pod1.yaml**

[root@k8s-master01 affi]# **kubectl create -f pod1.yaml**

pod/affinity created

[root@k8s-master01 affi]# **kubectl get pod -o wide**

NAME READY STATUS RESTARTS

AGE IP NODE NOMINATED NODE READINESS GATES

affinity 1/1 Running 0 42s

10.244.2.74 **k8s-node01** <none> <none>

[root@k8s-master01 affi]# **vim pod1.yaml**

[root@k8s-master01 affi]# **kubectl delete pod affinity**

pod "affinity" deleted

kube[root@k8s-master01 affi]**# kubectl create -f pod1.yaml**

pod/affinity created

[root@k8s-master01 affi]# **kubectl get pod -o wide**

NAME READY STATUS RESTARTS

AGE IP NODE NOMINATED NODE READINESS GATES

affinity 1/1 Running 0 17s

10.244.1.100 **k8s-node02** <none> <none>

[root@k8s-master01 affi]# **vim pod1.yaml**

[root@k8s-master01 affi]# **kubectl delete pod affinity**

pod "affinity" deleted

[root@k8s-master01 affi]# **kubectl create -f pod1.yaml**

pod/affinity created

[root@k8s-master01 affi]# **kubectl get pod -o wide**

NAME READY STATUS RESTARTS

AGE IP NODE NOMINATED NODE READINESS GATES

affinity 0/1 Pending 0 7s

<none> <none> <none> <none>

没满足硬策略就一直 pending

[root@k8s-master01 affi]# **vim pod2.yaml**

[root@k8s-master01 affi]# **kubectl delete pod affinity**

pod "affinity" deleted

[root@k8s-master01 affi]# **kubectl apply -f pod2.yaml**

pod/affinity created
[root@k8s-master01 affi]# **kubectl get pod**
NAME READY STATUS RESTARTS
AGE
affinity 1/1 Running 0 5s
[root@k8s-master01 affi]# **vim pod1.yaml**
[root@k8s-master01 affi]# **kubectl delete pod --all**
pod "affinity" deleted
pod "frontend-bw4zm" deleted
pod "frontend-n9qjp" deleted
pod "frontend-wl56d" deleted
pod "my-nginx-856cb6c947-6zc2h" deleted
pod "pod-deployment-57cf4db6cc-9fgmb" deleted
pod "pod-deployment-57cf4db6cc-ptqd9" deleted
pod "pod-deployment22-7958ddcc9d-bdpbf" deleted
pod "pod-deployment22-7958ddcc9d-krb2q" deleted
pod "test-pd" deleted
pod "test-pd1" deleted
pod "web-0" deleted
kubect^H[root@k8s-master01 affi]# **kubectl create -f pod1.yaml**
pod/node01 created
[root@k8s-master01 affi]# **kubectl get pod -o wid**e
NAME READY STATUS RESTARTS
node01 1/1 Running 0 8s
10.244.1.106 k8s-node02 <none> <none>
[root@k8s-master01 affi]# **vim pod3.yaml**
[root@k8s-master01 affi]# **kubectl create -f pod3.yaml**
pod/pod-3 created
[root@k8s-master01 affi]# **kubectl get pod**
NAME READY STATUS RESTARTS
AGE
node01 1/1 Running 0
5m52s
pod-3 1/1 Running 0 9s

[root@k8s-master01 affi]# **kubectl get pod -o wide**
NAME READY STATUS RESTARTS
AGE IP NODE NOMINATED NODE READINESS GATES
node01 1/1 Running 0 7s
10.244.1.109 k8s-node02 <none> <none>
pod-3 0/1 Pending 0 15m
<none> <none> <none> <none>
因为 **pod-3** 硬策略要求 **node01** 的 **app=node02,**所以不满足，只需把 **node01** 的标签改掉
[root@k8s-master01 affi]# **kubectl label pod node01 app=node02 --overwrite=true**

pod/node01 labeled
[root@k8s-master01 affi]# **kubectl get pod -o wide**
NAME READY STATUS RESTARTS
AGE IP NODE NOMINATED NODE READINESS GATES
node01 1/1 Running 0 66s
10.244.1.109 k8s-node02 <none> <none>
pod-3 1/1 Running 0 16m
10.244.1.110 k8s-node02 <none> <none>

[root@k8s-master01 ~]# **kubectl get node --show-labels**
NAME STATUS ROLES AGE VERSION LABELS
k8s-master01 Ready master 4d v1.15.1
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubern
etes.io/hostname=k8s-master01,kubernetes.io/os=linux,node-role.kubernetes.io/master=
k8s-node01 Ready <none> 3d22h v1.15.1
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubern
etes.io/hostname=k8s-node01,kubernetes.io/os=linux
k8s-node02 Ready <none> 3d22h v1.15.1
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubern
etes.io/hostname=k8s-node02,kubernetes.io/os=linux

同一个 node 可能在不同的 disk
我们改成 kubernetes.io/hostname,则会在同一个 disk

污点

1.1 污点查看

[root@k8s-master01 ~]# **kubectl describe node k8s-master01**
Name: k8s-master01
Roles: master
...
Taints: node-role.kubernetes.io/master**:NoSchedule**
...
[root@k8s-master01 affi]# **vim pod1.yaml**
[root@k8s-master01 affi]# **kubectl create -f pod1.yaml**

pod/node01 created

[root@k8s-master01 affi]# **kubectl get pod -o wide**

1.2 污点设置

[root@k8s-master01 ~]# **kubectl taint nodes k8s-node01 check=youyou:NoExecute**

node/k8s-node01 tainted

[root@k8s-master01 ~]# **kubectl get pod -o wide**

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

frontend-bxtpw 0/1 ImagePullBackOff 0 89m 10.244.1.102 k8s-**node02** <none> <none>

frontend-kcw62 0/1 ContainerCreating 0 12s <none> k8s-**node0**2 <none> <none>

frontend-kwsgv 0/1 ContainerCreating 0 12s <none> k8s-**node02** <none> <none>

my-nginx-856cb6c947-2drnl 1/1 Running 0 89m 10.244.1.104 k8s-node02 <none> <none>

node01 1/1 Running 0 57m 10.244.1.109 k8s-**node02** <none> <none>

pod-3 1/1 Running 0 73m

全都在 **node02** 上运行

[root@k8s-master01 ~]# **kubectl taint nodes k8s-node02 check=youyou:NoExecute**

node/k8s-node02 tainted

[root@k8s-master01 ~]# **kubectl get pod -o wide**

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

frontend-9qxld 0/1 Pending 0 20s <none> <none> <none> <none>

frontend-k2hjn 0/1 Pending 0 20s <none> <none> <none> <none>

frontend-rz6r8 0/1 Pending 0 20s <none> <none> <none> <none>

my-nginx-856cb6c947-4xp4c 0/1 Pending 0 20s <none> <none> <none> <none>

全部 **pending** 因为所有 **node** 都有污点

[root@k8s-master01 affi]# **vim pod3.yaml**

[root@k8s-master01 affi]# **kubectl delete pod pod-3**

pod/pod-3 created

[root@k8s-master01 affi]# **kubectl get pod -o wide**

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

frontend-9qxld 0/1 Pending 0 12m <none> <none> <none> <none>

frontend-k2hjn 0/1 Pending 0 12m <none> <none> <none> <none>

frontend-rz6r8 0/1 Pending 0 12m <none>
<none> <none> <none>
my-nginx-856cb6c947-4xp4c 0/1 Pending 0 12m <none>
<none> <none> <none>
pod-3 1/1 Running 0 7s
10.244.1.118 k8s-node02 <none> <none>
[root@k8s-master01 affi]# **kubectl taint nodes k8s-master01 node-role.kubernetes.io/master=:PreferNoSchedule**
node/k8s-master01 tainted

2.1 污点的删除

[root@k8s-master01 affi]# **history |grep taint**
767 kubectl taint nodes k8s-node01 check=youyou:NoExecute
769 kubectl taint nodes k8s-node02 check=youyou:NoExecute
789 kubectl taint nodes k8s-master01 node-role.kubernetes.io/master=:PreferNoSchedul
790 kubectl taint nodes k8s-master01 node-role.kubernetes.io/master=:PreferNoSchedule
793 history |grep taint
[root@k8s-master01 affi]# **kubectl taint nodes k8s-node01 check=youyou:NoExecute-**
node/k8s-node01 untainted
[root@k8s-master01 affi]# **kubectl taint nodes k8s-node02 check=youyou:NoExecute-**
node/k8s-node02 untainted
[root@k8s-master01 affi]# **kubectl describe node k8s-node01**
Name: k8s-node01
...
Taints: <none>

指定

固定节点因为比较简单粗暴，所以直接上操作。分为两类，一类是指定单个 node，另一类是指定 node 的 label。
[root@k8s-master01 affi]# **kubectl get pod -o wide**
NAME READY STATUS RESTARTS AGE IP
NODE NOMINATED NODE READINESS GATES
frontend-9qxld 0/1 Pending 0 16m <none>
<none> <none> <none>
frontend-k2hjn 0/1 Pending 0 16m <none>
<none> <none> <none>
frontend-rz6r8 0/1 Pending 0 16m <none>
<none> <none> <none>
my-nginx-856cb6c947-4xp4c 0/1 Pending 0 16m <none>
<none> <none> <none>
pod-3 1/1 Running 0 4m17s
10.244.1.118 k8s-node02 <none> <none>
[root@k8s-master01 node]# **kubectl apply -f pod2.yaml** （加一个 **disk=ssd)**标签

deployment.extensions/myweb1111 created

[root@k8s-master01 node]# **kubectl get pod**

NAME READY STATUS RESTARTS AGE

frontend-fbmt7 0/1 ErrImagePull 0 32s

frontend-rvj62 0/1 ErrImagePull 0 32s

frontend-tpl8k 0/1 ErrImagePull 0 32s

myweb1111-68f5d564c4-6gq9p 0/1 Pending 0 6s

myweb1111-68f5d564c4-czv2t 0/1 Pending 0 6s

web-0 0/1 ContainerCreating 0 25s

[root@k8s-master01 node]# **kubectl label node k8s-node01 disk=ssd**

node/k8s-node01 labeled(给 **node** 加标签使其满足）

[root@k8s-master01 node]# **kubectl get pod -o wide**

NAME READY STATUS RESTARTS AGE

IP NODE NOMINATED NODE READINESS GATES

myweb1111-68f5d564c4-6gq9p 1/1 Running 0 111s

10.244.2.98 k8s-node01 <none> <none>

myweb1111-68f5d564c4-czv2t 1/1 Running 0 111s

10.244.2.99 k8s-node01 <none> <none>

[root@k8s-master01 node]# **kubectl label node k8s-node02 disk=ssd**

node/k8s-node02 labeled

[root@k8s-master01 node]# **kubectl get deployment**

NAME READY UP-TO-DATE AVAILABLE AGE

myweb1111 2/2 2 2 2m34s

[root@k8s-master01 node]# **kubectl edit deployment myweb1111**

deployment.extensions/myweb1111 edited

[root@k8s-master01 node]# **kubectl get deployment**

NAME READY UP-TO-DATE AVAILABLE AGE

myweb1111 8/8 8 8 3m51s

[root@k8s-master01 node]# **kubectl get pod -o wide**

NAME READY STATUS RESTARTS AGE

IP NODE NOMINATED NODE READINESS GATES

myweb1111-68f5d564c4-5cg7r 1/1 Running 0 8s

10.244.1.124 k8s-node02 <none> <none>

myweb1111-68f5d564c4-6gq9p 1/1 Running 0 3m55s

10.244.2.98 k8s-node01 <none> <none>

myweb1111-68f5d564c4-6jd4t 1/1 Running 0 8s

10.244.1.121 k8s-node02 <none> <none>

myweb1111-68f5d564c4-czv2t 1/1 Running 0 3m55s

10.244.2.99 k8s-node01 <none> <none>

myweb1111-68f5d564c4-k6lzp 1/1 Running 0 8s

10.244.1.122 k8s-node02 <none> <none>

myweb1111-68f5d564c4-n8srw 1/1 Running 0 8s

10.244.2.101 k8s-node01 <none> <none>

myweb1111-68f5d564c4-ng7lk 1/1 Running 0 8s

10.244.2.100 k8s-node01 <none> <none>
myweb1111-68f5d564c4-pqknj 1/1 Running 0 8s
10.244.1.123 k8s-node02 <none> <none>
web-0 0/1 ContainerCreating 0 4m14s
<none> k8s-node02 <none> <none>

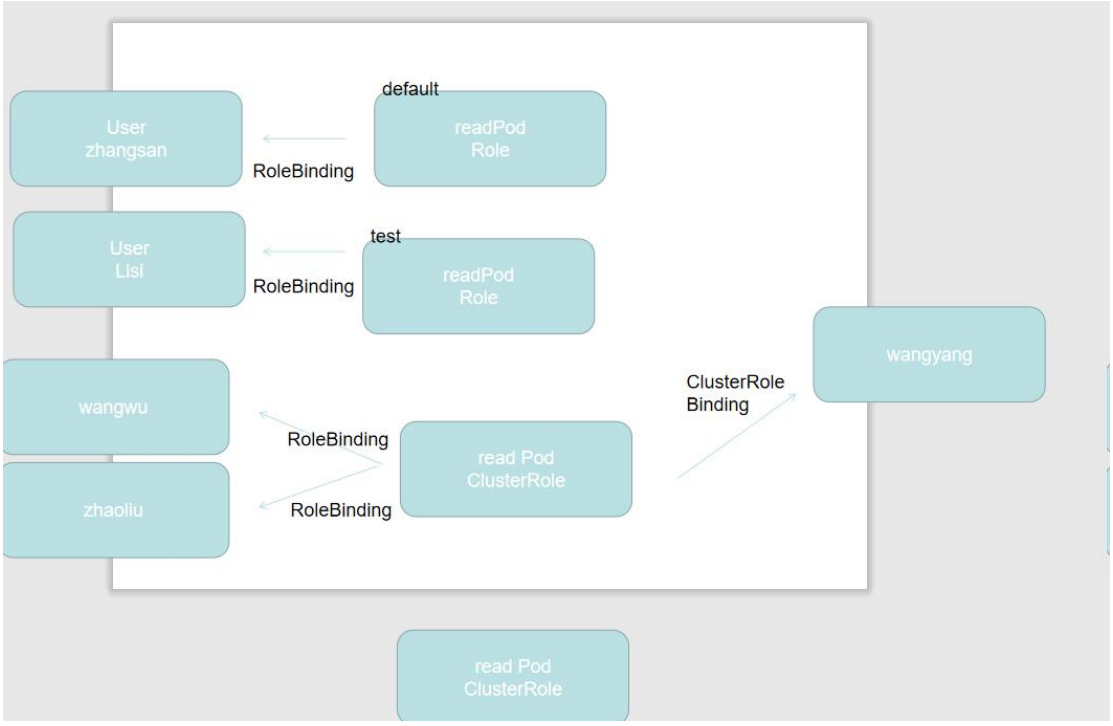
RBAC

RBAC，全称 Role-based access control，会创建很多 Role，每个 Role 对应着一系列的权限。通过将用户和 Role 进行绑定来给用户赋予不同类型的权限。

Role 和 ClusterRole
K8s 中的资源要么是在某个 namespace 下，例如 pod 和 service，要么是全局的，例如 node。所有权限也就分为两种，Role 下的权限是针对某个 namespace 的，在定义的时候必须要加上具体 namespace；与之相对的 ClusterRole 下的权限是集群级别的，不用加 namespace。

需要注意，所有的权限都是允许类型，而没有拒绝类型，所以赋予权限只能进行添加操作。一个用户绑定了某个 Role，就只有 Role 下赋予的权限。

这里面带 system: 前缀的是系统级别的，不建议修改和使用，而面向用户的，例如 cluster-admin，admin，edit，view 等则可以使用和修改，接下来的例子会看到。

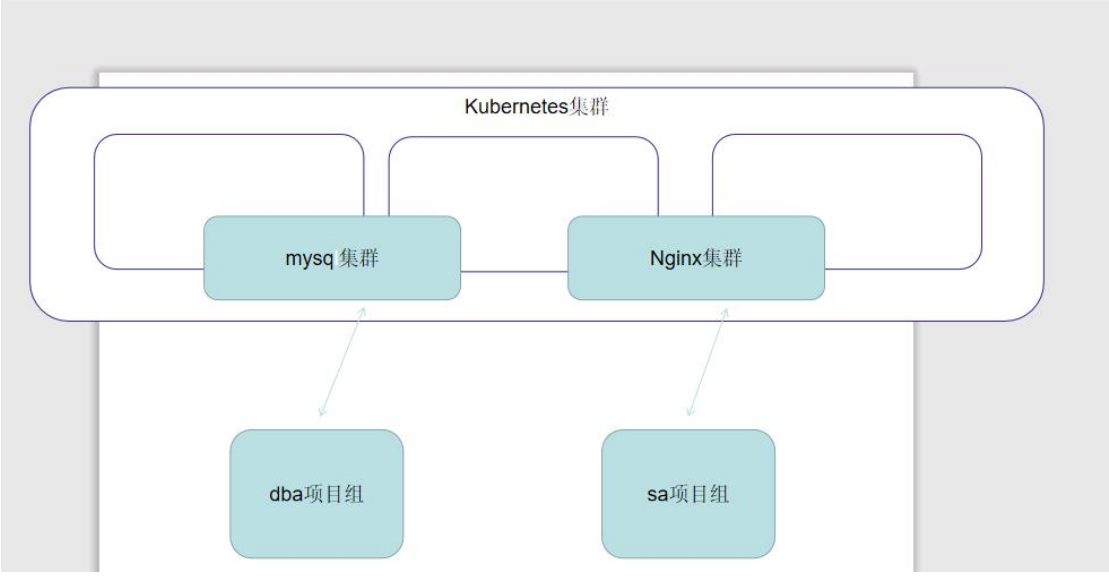


名称空间级别
不同名称空间级别有同一个动作，分别弄会浪费资源，麻烦
所以都通过 rolebing 链接 clustrole，只能获取某些权限
而集群管理者通过 kusterrolebing 链接 clusterrole，可以获得所有权限

config 文件

用户在访问 k8s 集群的时候需要携带自己的的认证信息，例如私钥和证书。于是 kuberctl 工具规定用户需要将这些信息连同 API server 的访问地址还有集群域名等等一起汇总填写到 ~/.kube/config 文件中，每次用户使用 kubectl 进行操作都会使用该文件进行认证。所以创建用户的根本目的就是创建该用户的 config 文件。

同时不管是集群内的还是集群外机器安装的 kubectl，只要能读取到用户的 config 文件就可以正常访问到集群资源。



资源分割

总结

k8s 将一系列权限以 Role 或者 ClusterRole 的方式划分角色
通过将用户与角色进行绑定达到为用户分配权限的目的
k8s 本身不保存用户信息，每次用户访问必须要通过自己的 config 文件携带认证信息
创建用户的 config 文件分为如下几步
构建证书请求 json 文件 csr.json
结合 CA 证书生成用户的客户端证书和私钥
创建空的 config 文件
补充 config 文件 cluster 信息和用户信息以及 context
将 config 文件复制到~/.kube/config 并设定好权限
集群外部机器想访问集群资源，复制用户 config 文件即可

如果是手动安装一套 k8s 应用出来，我们需要分别创建应用中各个组件的 Deployment 以及 Service 的 yaml 文件。如果想在另一个 k8s 集群中去部署相同的一套应用，其实只需要将相同的一套 yaml 文件拷贝过去即可，顶多在更改几个参数。那么完全可以将应用所需要的 yaml 文件打包放在一个仓库里直接供人下载使用即可，这就是 Helm 的初衷。

不过当然 Helm 还集成了很多别的功能，例如一键更新和回滚，yaml 文件设置动态参数等等，后面我们会慢慢学习到。

在真正开始使用 Helm 前有几个特有的概念要先熟悉下。

Chart

Chart 就是按照一定目录结构保存的多个文件，用来描述所部署应用相关 k8s 资源的信息。Chart 每被 Helm 使用一次，就部署一个应用到集群中，如果 Chart 被使用多次，就会把相同的应用部署多次。

通常是直接下载官方的 Chart，微调下参数然后使用。也可以跟着官方文档中的步骤来创建自己的 Chart，下面的实际操作中我们两种方式都会演示。

Release

可以将 Helm 的 Chart 类比 Docker 的 Image，那么 Release 就是 Docker 的 Container。一个 Chart 可以生成多个 Release。

Repo

Repo 就是存放 Chart 的仓库，和 Docker Hub 一样，Helm 也有自己的 Helm Hub。当然也是可以建立自己的私有仓库的。

安装 Helm

[root@k8s-master01 ~]# **tar -zxvf helm-v3.3.0-rc.2-linux-amd64.tar.gz**
linux-amd64/
linux-amd64/helm
linux-amd64/LICENSE
linux-amd64/README.md
[root@k8s-master01 ~]# **cd linux-amd64/**
[root@k8s-master01 linux-amd64]# **cp helm /usr/local/bin 将二进制文件加入 PATH 即可**

使用官方 Chart

1.首先添加官方的 Repo

[root@k8s-master01 linux-amd64]# **helm repo add stable https://kubernetes.oss-cn-hangzhou.aliyuncs.com/charts**

"stable" has been added to your repositories

2.搜索 repo 中的 chart

[root@k8s-master01 linux-amd64]# **helm search repo stable**

NAME	CHART VERSION	APP VERSION	DESCRIPTION
stable/acs-engine-autoscaler	2.1.3	2.1.1	Scales worker nodes within agent pools
stable/aerospike	0.1.7	v3.14.1.2	A Helm chart for Aerospike in Kubernetes
stable/anchore-engine	0.1.3	0.1.6	Anchore container analysis and policy evaluatio...
stable/artifactory	7.0.3	5.8.4	Universal Repository Manager supporting all maj...
stable/artifactory-ha	0.1.0	5.8.4	Universal Repository Manager supporting all maj...
stable/aws-cluster-autoscaler	0.3.2		Scales worker nodes within autoscaling groups.
stable/bitcoind	0.1.0	0.15.1	Bitcoin is an innovative payment network and a ...
stable/buildkite	0.2.1	3	Agent for Buildkite

...一堆

想要进行模糊搜索的话可以用 **helm search hub xxx**

如果发现是 **stable** 下的就可以去 **stable** 中安装了。

接下来就可以安装 **chart** 了，命令是 **helm install name chart**，不过这里的 **chart** 可以是下面 **5** 中来源的一个

By chart reference: helm install mymaria example/mariadb
By path to a packaged chart: helm install mynginx ./nginx-1.2.3.tgz
By path to an unpacked chart directory: helm install mynginx ./nginx
By absolute URL: helm install mynginx https://example.com/charts/nginx-1.2.3.tgz
By chart reference and repo url: helm install --repo https://example.com/charts/ mynginx nginx

用官方的 stable 来安装 chart

[root@k8s-master01 linux-amd64]# **helm repo update**

Hang tight while we grab the latest from your chart repositories...

...Successfully got an update from the "stable" chart repository
Update Complete. ⎈ Happy Helming!⎈
[root@k8s-master01 linux-amd64]# **helm install stable/mysql --generate-name**
NAME: mysql-1596763060
LAST DEPLOYED: Fri Aug 7 09:17:40 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
...
[root@k8s-master01 linux-amd64]# **helm list**
NAME NAMESPACE REVISION UPDATED
STATUS CHART APP VERSION
mysql-1596763060 default 1 2020-08-07 09:17:40.774039558 +0800 CST
deployed mysql-0.3.5
[root@k8s-master01 linux-amd64]# **helm status mysql-1596763060**
NAME: mysql-1596763060
LAST DEPLOYED: Fri Aug 7 09:17:40 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
...
[root@k8s-master01 linux-amd64]# **helm pull stable/mysql**
[root@k8s-master01 linux-amd64]# **ls**
helm LICENSE mysql-0.3.5.tgz README.md
[root@k8s-master01 linux-amd64]# **tar -zxvf mysql-0.3.5.tgz**
mysql/Chart.yaml
tar: mysql/Chart.yaml：不可信的旧时间戳 1970-01-01 08:00:00
mysql/values.yaml
tar: mysql/values.yaml：不可信的旧时间戳 1970-01-01 08:00:00
mysql/templates/NOTES.txt
tar: mysql/templates/NOTES.txt：不可信的旧时间戳 1970-01-01 08:00:00
mysql/templates/_helpers.tpl
tar: mysql/templates/_helpers.tpl：不可信的旧时间戳 1970-01-01 08:00:00
mysql/templates/configmap.yaml
tar: mysql/templates/configmap.yaml：不可信的旧时间戳 1970-01-01 08:00:00
mysql/templates/deployment.yaml
tar: mysql/templates/deployment.yaml：不可信的旧时间戳 1970-01-01 08:00:00
mysql/templates/pvc.yaml
tar: mysql/templates/pvc.yaml：不可信的旧时间戳 1970-01-01 08:00:00
mysql/templates/secrets.yaml

tar: mysql/templates/secrets.yaml：不可信的旧时间戳 1970-01-01 08:00:00
mysql/templates/svc.yaml
tar: mysql/templates/svc.yaml：不可信的旧时间戳 1970-01-01 08:00:00
mysql/.helmignore
tar: mysql/.helmignore：不可信的旧时间戳 1970-01-01 08:00:00
mysql/README.md
tar: mysql/README.md：不可信的旧时间戳 1970-01-01 08:00:00
[root@k8s-master01 linux-amd64]# **cd mysql**
[root@k8s-master01 mysql]# **ll**
总用量 16
-rwxr-xr-x 1 root root 420 1 月 1 1970 Chart.yaml
-rwxr-xr-x 1 root root 7366 1 月 1 1970 README.md
drwxr-xr-x 2 root root 140 8 月 7 09:23 templates
-rwxr-xr-x 1 root root 1970 1 月 1 1970 values.yaml
[root@k8s-master01 mysql]# **helm uninstall mysql-1596763060**
release "mysql-1596763060" uninstalled

创建本地 chart

- 文件结构

```
[root@master mychart]# tree mychart/
mychart/
├── charts
├── Chart.yaml
├── templates
│   ├── deployment.yaml                    # 部署相关资源
│   ├── _helpers.tpl                           # 模版助手
│   ├── ingress.yaml                           # ingress资源
│   ├── NOTES.txt                                  # chart的帮助文本，运行helm install时
│   ├── service.yaml                           # service端点
│   └── tests
│       └── test-connection.yaml
└── values.yaml

3 directories, 8 files
```

1.创建一个 Chart.yaml 文件如下

```
apiVersion: v1
name: youyou
version: 1.0.0
```

2. chart 文件夹

用来放依赖的 chart

3. templates 目录

deployment.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: mynginx-deployment
spec:
  replicas: {{ .Values.replica }}
  template:
    metadata:
      labels:
        app: mynginx
        version: v2
    spec:
      containers:
        - name: mynginx
          image: {{ .Values.image }}:{{ .Values.imageTag }}
          ports:
            - containerPort: 80
```

Service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: mynginx-service
  namespace: default
spec:
  type: NodePort
  selector:
    app: mynginx
    version: v2
  ports:
    - name: http
      port: 8080
      targetPort: 80
      nodePort: 30000
```

4. values.yaml

```
image: hub.atguigu.com/library/mynew
imageTag: v1
replica: 5
```

~

[root@k8s-master01 mysql]# **mkdir youyou**
[root@k8s-master01 mysql]# **cd youyou**
[root@k8s-master01 youyou]# **vim Chart.yaml**
[root@k8s-master01 youyou]# **mkdir templates**
[root@k8s-master01 youyou]#**cd templates**
[root@k8s-master01 templates]# **vim deployment.yaml**
[root@k8s-master01 templates]# **vim service.yaml**
[root@k8s-master01 youyou]# **cd ..**
[root@k8s-master01 youyou]# **vim values.yaml**
[root@k8s-master01 youyou]# **cd ..**
[root@k8s-master01 mysql]# ls
Chart.yaml README.md templates values.yaml youyou
[root@k8s-master01 mysql]# **mv youyou ../**
[root@k8s-master01 mysql]# **cd ..**
[root@k8s-master01 linux-amd64]# **ls**
helm LICENSE mysql mysql-0.3.5.tgz README.md youyou
[root@k8s-master01 linux-amd64]# **helm install youyou --generate-name**
NAME: youyou-1596764558
LAST DEPLOYED: Fri Aug 7 09:42:38 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
[root@k8s-master01 linux-amd64]# **kubectl get pod**
[root@k8s-master01 linux-amd64]# cd youyou
[root@k8s-master01 youyou]# cd templates/
[root@k8s-master01 templates]# vim deployment.yaml
[root@k8s-master01 templates]# cd ../../
[root@k8s-master01 linux-amd64]# **helm install youyou --generate-name**
Error: rendered manifests contain a resource that already exists. Unable to continue with install: Service "mynginx-service" in namespace "default" exists and cannot be imported into the current release: invalid ownership metadata; annotation validation error: key "meta.helm.sh/release-name" must equal "youyou-1596764769": current value is "youyou-1596764558"
[root@k8s-master01 linux-amd64]# **helm list**
NAME NAMESPACE REVISION UPDATED
STATUS CHART APP VERSION
youyou-1596764558 default 1 2020-08-07 09:42:38.934030246 +0800 CST
deployed youyou-1.0.0
[root@k8s-master01 linux-amd64]# **helm uninstall youyou-1596764558**
release "youyou-1596764558" uninstalled（上一次失败的卸载掉）
[root@k8s-master01 linux-amd64]# **helm install youyou --generate-name**
NAME: youyou-1596764976
LAST DEPLOYED: Fri Aug 7 09:49:37 2020

NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
[root@k8s-master01 linux-amd64]# **kubectl get pod**
NAME READY STATUS RESTARTS AGE
mynginx-deployment-745fb5446b-qj9n2 1/1 Running 0 18s
mynginx-deployment-745fb5446b-rxbqf 1/1 Running 0 18s
mynginx-deployment-745fb5446b-tdfdp 1/1 Running 0 18s
[root@k8s-master01 linux-amd64]# **kubectl get svc**
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 4d16h
mynginx-service NodePort 10.100.129.148 <none> 8080:30000/TCP 29s

修改

[root@k8s-master01 linux-amd64]# **cd youyou/templates**
[root@k8s-master01 templates]# **vim deployment.yaml**
[root@k8s-master01 templates]# **cd ..**
[root@k8s-master01 youyou]# **vim values.yaml**
[root@k8s-master01 youyou]# **helm list**
NAME NAMESPACE REVISION UPDATED STATUS CHART APP VERSION
youyou-1596764976 default 1 2020-08-07 09:49:37.043645471 +0800 CST deployed youyou-1.0.0
[root@k8s-master01 youyou]# **helm upgrade youyou-1596764976**
Error: "helm upgrade" requires 2 arguments

Usage: helm upgrade [RELEASE] [CHART] [flags]
[root@k8s-master01 youyou]# helm upgrade youyou-1596764976 .
Release "youyou-1596764976" has been upgraded. Happy Helming!
NAME: youyou-1596764976
LAST DEPLOYED: Fri Aug 7 09:57:11 2020
NAMESPACE: default
STATUS: deployed
REVISION: 2
TEST SUITE: None
[root@k8s-master01 youyou]# **kubectl get pod**
NAME READY STATUS RESTARTS AGE
mynginx-deployment-745fb5446b-qj9n2 1/1 Running 0

```
7m44s
mynginx-deployment-745fb5446b-rxbqf   1/1     Running   0
7m44s
mynginx-deployment-745fb5446b-tdfdp   1/1     Running   0
7m44s
mynginx-deployment-745fb5446b-xtqhc   1/1     Running   0          10s
[root@k8s-master01 youyou]# kubectl get svc
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
kubernetes        ClusterIP   10.96.0.1        <none>        443/TCP
4d16h
mynginx-service   NodePort    10.100.129.148   <none>        8080:30000/TCP
7m56s
[root@k8s-master01 youyou]# helm history youyou-1596764976
REVISION  UPDATED                   STATUS      CHART          APP      VERSION
      DESCRIPTION
1         Fri Aug  7 09:49:37 2020  superseded  youyou-1.0.0             Install
complete
2         Fri Aug  7 09:57:11 2020  deployed    youyou-1.0.0             Upgrade
complete
[root@k8s-master01 youyou]# helm rollback youyou-1596764976
Rollback was a success! Happy Helming!
```

就跟 Deployment 的回滚道理一样，因为旧的 pod 都没有被彻底删除，只是停用，所以回滚会很快。

这里只是简单的演示，更多的功能可以参考官方文档以及 helm help 帮助文档说明。

Helm 常用命令汇总
把上面用到的一些常用命令汇总以下

命令	说明
helm search hub xxx	在 Helm Hub 上搜索 Chart
helm search repo repo_name	在本地配置的 Repo 中搜索 Chart
helm install release_name chart_reference	chart 一共有 5 种 reference
helm list	查看已部署的 release
helm status release_name	查看 release 信息
helm upgrade release_name chart_reference	修改 chart 信息后升级 release
helm history release_name	查看 release 的更新历史记录
helm rollback release_name revision	回滚操作
helm uninstall release_name	卸载 release