

第0章 参考网站

```
1 0.k8s介绍  
2 https://kubernetes.io/zh/docs/concepts/overview/what-is-kubernetes/  
3  
4 1.docker官方文档  
5 https://docs.docker.com/  
6  
7 2.k8s官方文档  
8 https://kubernetes.io/zh/  
9  
10 3.kubeadm官方文档  
11 https://kubernetes.io/zh/docs/setup/production-environment/tools/  
12  
13 4.prometheus官方文档  
14 https://prometheus.io/docs/introduction/overview/  
15  
16 5.ansible安装k8s项目  
17 https://github.com/easylab/kubeasz  
18  
19 6.阿里云ACK  
20 https://www.aliyun.com/product/kubernetes  
21  
22 7.黑科技-好用的API查询网站  
23 https://k8s.mybatis.io/v1.19/  
24  
25 8.黑科技-自动生成yaml资源清单  
26 https://k8syaml.com/
```

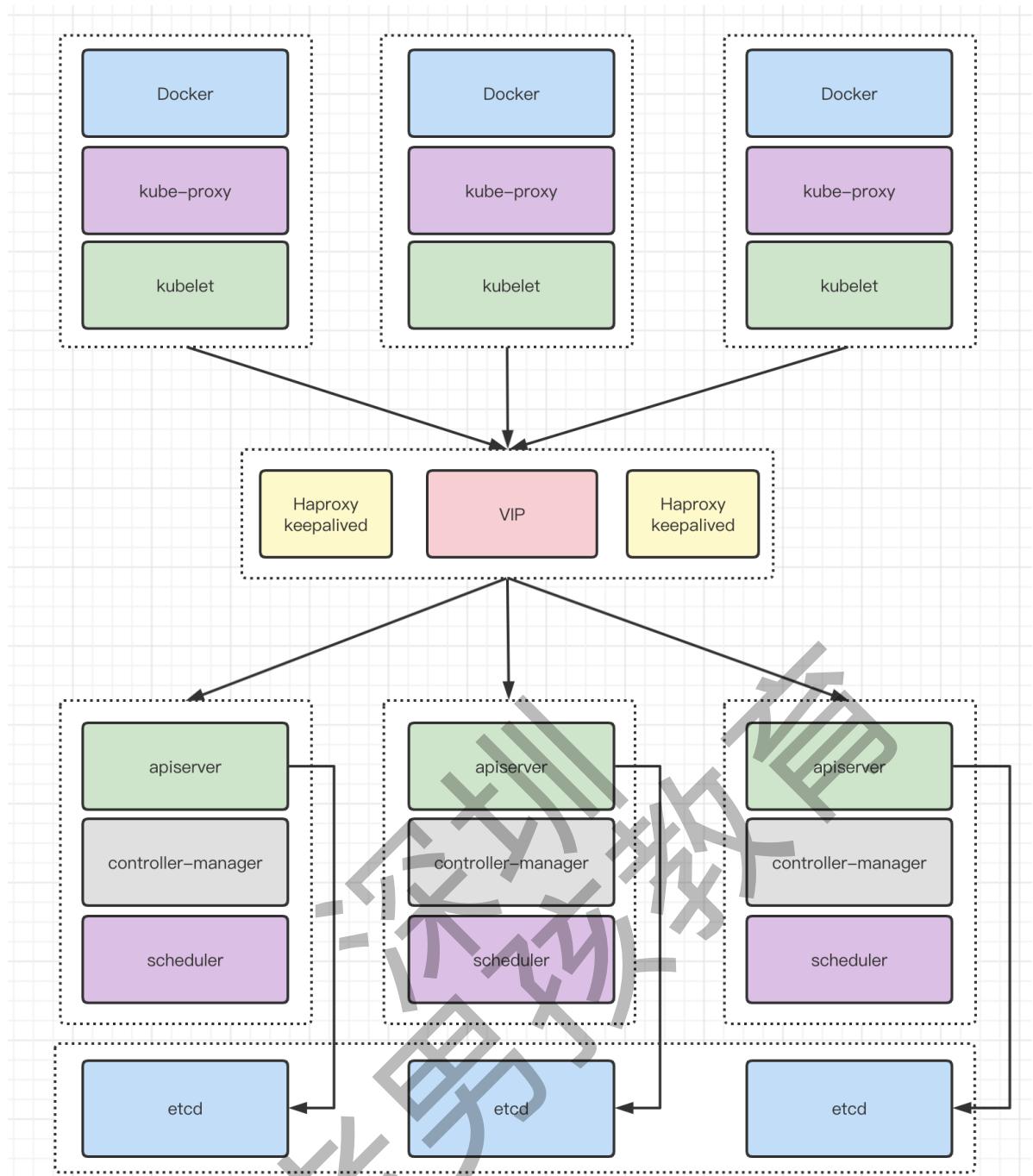
第1章 k8s介绍

<https://kubernetes.io/zh/docs/concepts/overview/what-is-kubernetes/>

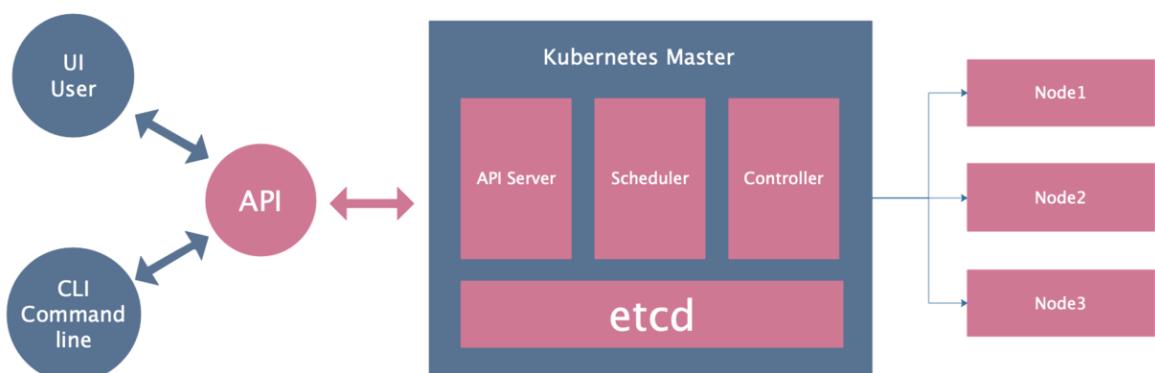
第2章 k8s系统架构和组件介绍

1.k8s架构

```
1 从系统架构来看，k8s分为2个节点  
2 Master 控制节点 指挥官  
3 Node 工作节点 干活的
```



2.Master节点组件



kube-API Server

- 1 kube-API Server: 提供k8s API接口
- 2 主要处理Rest操作以及更新Etcd中的对象
- 3 是所有资源增删改查的唯一入口。

Scheduler

- 1 Scheduler: 资源调度器
- 2 根据etcd里的节点资源状态决定将Pod绑定到哪个Node上

Controller Manager

- 1 Controller Manager
- 2 负责保障pod的健康存在
- 3 资源对象的自动化控制中心，Kubernetes集群有很多控制器。

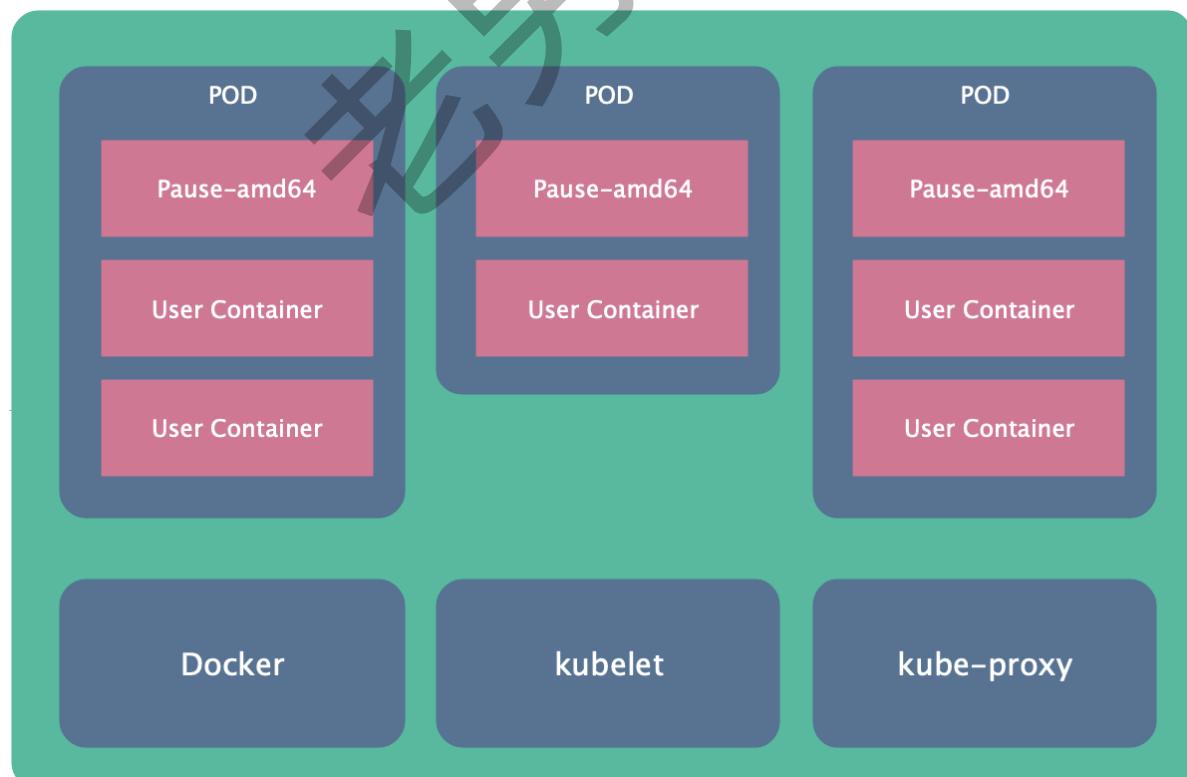
Etcd

- 1 这个是Kubernetes集群的数据库
- 2 所有持久化的状态信息存储在Etcd中

kubectl

- 1 kubectl是管理k8s集群的客户端工具，管理员通过kubectl命令对API Server进行操作
- 2 API Server 响应并返回对应的命令结果，从而达到对 Kubernetes 集群的管理

3.Node节点的组成



容器运行时 (Container Runtime)

- 1 容器运行时是负责容器运行的软件。
- 2 默认情况下，Kubernetes 使用 容器运行时接口（Container Runtime Interface, CRI） 来与你所选择的容器运行时交互。
- 3
- 4 如果同时检测到 Docker 和 containerd，则优先选择 Docker。这是必然的，因为 Docker 18.09 附带了 containerd 并且两者都是可以检测到的，即使你仅安装了 Docker。如果检测到其他两个或多个运行时，kubeadm 输出错误信息并退出。
- 5
- 6 kubelet 通过内置的 dockershim CRI 实现与 Docker 集成。

Docker Engine

- 1 负责节点容器的管理工作，最终创建出来的是一个Docker容器。

kubelet

- 1 安装在Node上的代理服务，用来管理Pods以及容器/镜像/volume等，实现对集群对节点的管理。

kube-proxy

- 1 安装在Node上的网络代理服务，提供网络代理以及负载均衡，实现与Service通讯。

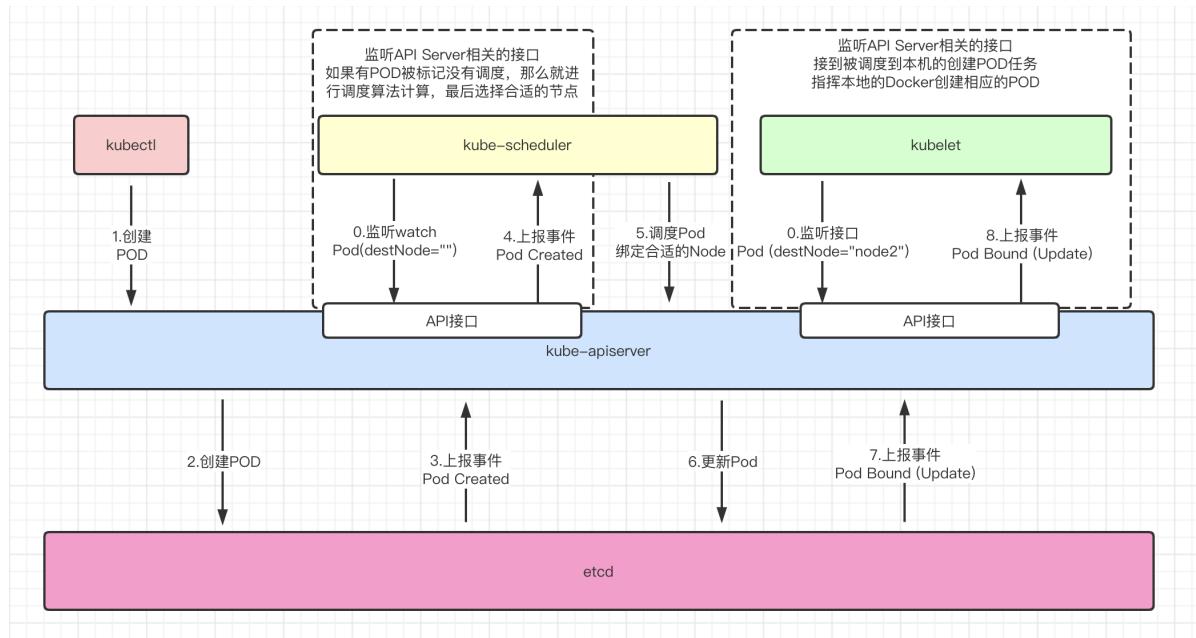
第3章 k8s核心资源

1.POD

- 1 POD是k8s的最小单位
- 2 POD的IP地址是随机的，删除POD会改变IP
- 3 POD都有一个根容器
- 4 一个POD内可以由一个或多个容器组成
- 5 一个POD内的容器共享根容器的网络命名空间和文件系统卷
- 6 一个POD的内的网络地址由根容器提供



创建Pod流程图:



文字说明：

- 1 创建Pod的清单 --> API Server
- 2 API Server --> etcd
- 3 etcd --> API Server
- 4 kube-scheduler --> API Server --> etcd
- 5 etcd --> API server --> kubelet --> Docker
- 6 kubelet --> API Server --> etcd
- 7
- 8 管理员清单提交给了 API Server
- 9 API Server 接收到请求后把资源清单信息存入到 etcd 数据库中
- 10 etcd 存好后更新状态给 API Server
- 11 API Server通过相应的接口更新事件
- 12 kube-scheduler 组件发现这个时候有一个 Pod 还没有绑定到节点上，就会对这个 Pod 进行一系列的调度，把它调度到一个最合适的节点上
- 13 然后把这个节点和 Pod 绑定到一起的信息告诉 API Server
- 14 API Server将调度消息更新到etcd里，etcd更新好后返回状态给API Server
- 15 节点上的 kubelet 组件这个时候 watch 到有一个 Pod 被分配过来了，就去把这个 Pod 的信息拉取下来，然后根据描述通过容器运行时把容器创建出来，最后当然同样把 Pod 状态汇报给API Server 再写回到 etcd 中去，这样就完成了一整个的创建流程。

容器运行状态：

- 1 waiting (等待)
- 2 如果容器并不处在Running或Terminated状态之一，它就处在Waiting状态。 处于Waiting状态的容器仍在运行它完成启动所需要的操作：例如，从某个容器镜像 仓库拉取容器镜像，或者向容器应用数据等等。 当你使用 kubectl 来查询包含 Waiting 状态的容器的 Pod 时，你也会看到一个 Reason 字段，其中给出了容器处于等待状态的原因。
- 3
- 4 Running (运行中)
- 5 Running状态表明容器正在执行状态并且没有问题发生。
- 6 如果你使用 kubectl 来查询包含 Running 状态的容器的 Pod 时，你也会看到 关于容器进入 Running状态的信息。
- 7
- 8 Terminated (已终止)
- 9 处于Terminated状态的容器已经开始执行并且或者正常结束或者因为某些原因失败。 如果你使用 kubectl来查询包含Terminated状态的容器的 Pod 时，你会看到 容器进入此状态的原因、退出代码以及容器执行期间的起止时间。

2.Label

- 1 Label标签是kubernetes中非常重要的一个属性，Label标签就像身份证一样，可以用来识别k8s的对象。
- 2 我们在传统架构里不同组件之间查找都是通过IP地址，但是在k8s里，很多的匹配关系都是通过标签来查找的。

3.Namespace

- 1 Namespace(命名空间)是k8s里另一个非常重要的概念，Namespace通过将集群内部的资源对象划分为不同的。Namespace里，形成逻辑上的不同项目组或用户组。
- 2 常见的pod,service,Deployment等资源如果没有指定命令空间，则默认创建在名为default的默认命名空间。

4.Controller

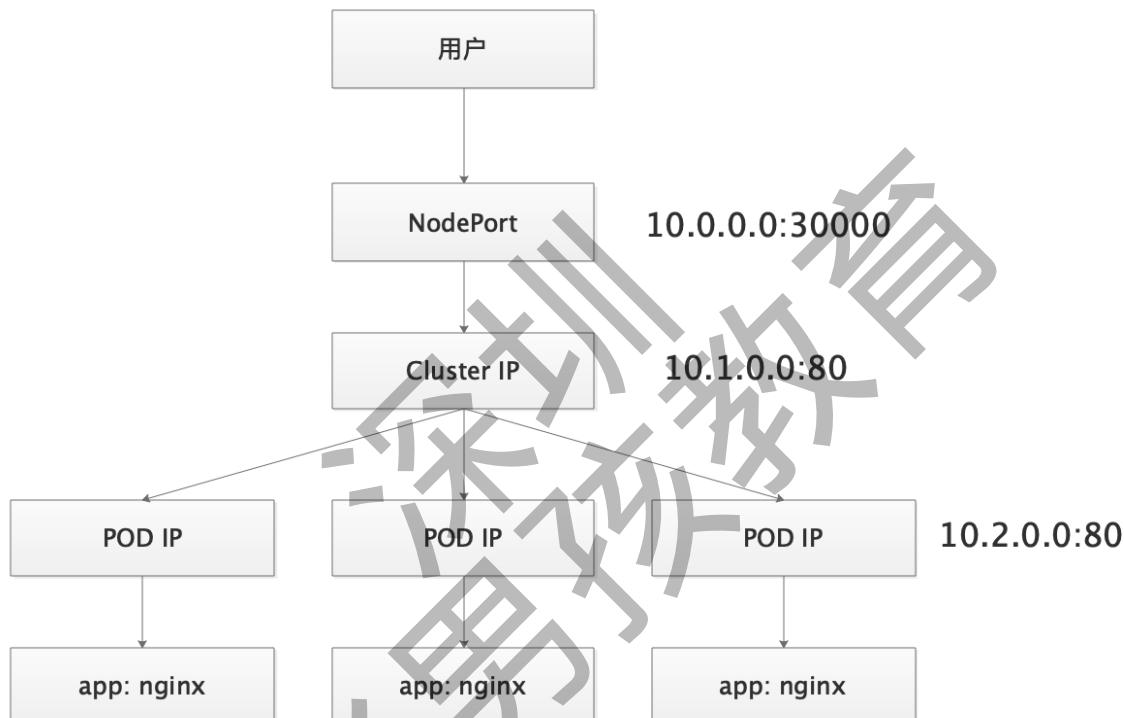
- 1 用来管理POD,控制器的种类有很多
 - RC Replication Controller 控制POD有多个副本
 - RS ReplicaSet RC控制的升级版
 - Deployment 推荐使用，功能更强大，包含了RS控制器
 - DaemonSet 保证所有的Node上有且只有一个Pod在运行
 - StatefulSet 有状态的应用，为Pod提供唯一的标识，它可以保证部署和scale的顺序

5.Service

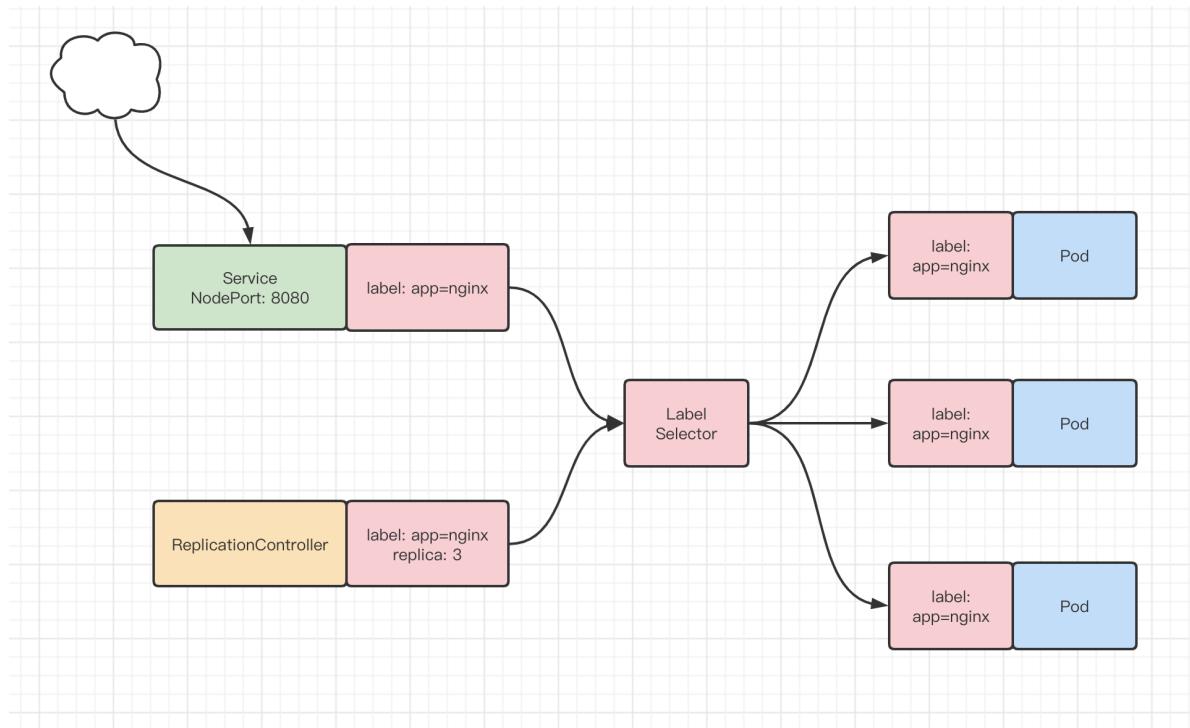
- 1 Service服务也是k8s的核心资源之一，Service定义了服务的入口地址，用来将后端的Pod服务暴露给外部用户访问。
- 2 我们知道Pod会随时被销毁和创建，而新创的Pod的IP地址并不是固定的，那么Service资源是如何找到后面不确定IP的Pod呢？
- 3
- 4 为了解决这个问题，k8s在PodIP之上又设计了一层IP，名为clusterIP，这个clusterIP在创建后就不会再变化，除非被删除重新创建。
- 5 通过这样的设计，我们只需要创建Cluster类型的Service资源，再通过标签和后端的Pod绑定在一起，这样只要访问clusterIP就可以将请求转发到了后端的Pod，而且如果后端有多个Pod，clusterIP还会将流量自动负载均衡到后面的Pod
- 6

7 通过ClusterIP我们可以访问对应的Pod，但是这个ClusterIP是一个虚拟IP，只能在k8s集群内部使用。外部用户是访问不到的，那么我们如何让外面的用户也可以访问到我们的服务呢？
8 我们的k8s运行在物理服务器上，那么必然会有物理网卡和真实的IP地址，也就是说所有k8s集群之外的节点想访问k8s集群内的某个节点，都必须通过物理网卡和IP。
9 所以我们可以通过将物理网卡的IP地址的端口和Cluster做端口映射来实现外部流量对内部Pod的访问。
10 这种IP在k8s里称为NodeIP，暴露的端口称为NodePort。
11
12
13 总结一下：
14 NodeIP 对外提供用户访问
15 ClusterIP 集群内部IP，可以动态感知后面的POD IP
16 POD IP POD的IP

Service三种网络图：



Service和Pod和RS的关系：



第4章 k8s实验环境准备

1.配置信息

	主机名	IP地址	推荐配置
1	master	10.0.0.10	1C2G40G
2	node1	10.0.0.11	1C2G40G
3	node2	10.0.0.12	1C2G40G

2.系统环境准备

配置主机名

	主机名	IP地址
1	master	10.0.0.10
2	node1	10.0.0.11
3	node2	10.0.0.12

配置host解析

```

1 cat >> /etc/hosts << EOF
2 10.0.0.10 master
3 10.0.0.11 node1
4 10.0.0.12 node2
5 EOF

```

关闭防火墙

```

1 systemctl stop firewalld NetworkManager
2 systemctl disable firewalld NetworkManager

```

关闭SELinux

```
1 setenforce 0
2 sed -i 's#SELINUX=disabled#SELINUX=disabled#g' /etc/selinux/config
3 getenforce
```

更新好阿里源

```
1 curl -o /etc/yum.repos.d/CentOS-Base.repo
http://mirrors.aliyun.com/repo/Centos-7.repo
2 curl -o /etc/yum.repos.d/epel.repo http://mirrors.aliyun.com/repo/epel-7.repo
3 sed -i '/aliyuncs/d' /etc/yum.repos.d/*.repo
```

确保网络通畅

```
1 ping -c 1 www.baidu.com
2 ping -c 1 master
3 ping -c 1 node1
4 ping -c 1 node2
```

配置时间同步

```
1 yum install chrony -y
2 systemctl start chronyd
3 systemctl enable chronyd
4 date
```

关闭SWAP分区

```
1 swapoff -a
2 sed -i '/swap/d' /etc/fstab
3 free -h
```

第5章 安装部署kubeadm

1.k8s安装部署方式

```
1 k8s的安装方式有很多种，常见的安装方式主要有以下几种：
2 1.二进制安装 生产推荐
3 2.kubeadm安装 学习和实验
4 3.网友写的ansible二进制安装 非常牛叉
5 4.第三方安装工具比如rancher
6 5.云服务的k8s 比如阿里云的ACK
7
8 这里我们应该把学习使用k8s作为首要任务，所以先选择安装最简单的kubeadm来部署一套k8s集群。
```

2.kubeadm介绍

```
1 kubeadm是google推出的一种部署k8s集群的套件，他将k8s的组件打包封装成了容器，然后通过
kubeadm进行集群的初始化。
2 kubeadm的好处是安装部署方便，缺点是不容易看到每个组件的配置，出了问题不好排查。但是我们现
在的重点是学习k8s的使用，所以这些优先选择简单的安装方式，最后会介绍二进制安装部署。
```

3.kubeadm部署前规划

节点规划

```
1 master master节点 API Server,controller,scheduler,kube-proxy,kubelet,etcd
2 node1 node节点 Docker kubelet kube-proxy
3 node2 node节点 Docker kubelet kube-proxy
```

IP规划

```
1 POD IP      10.2.0.0
2 Cluster IP  10.1.0.0
3 Node IP     10.0.0.0
```

注意！！！以下步骤如果没有特别指出在某个节点运行则默认在所有节点都执行。

4.kubeadm部署前系统环境准备

设置k8s禁止使用swap

```
1 cat > /etc/sysconfig/kubelet<<EOF
2 KUBELET_CGROUP_ARGS="--cgroup-driver=systemd"
3 KUBELET_EXTRA_ARGS="--fail-swap-on=false"
4 EOF
```

设置内核参数

```
1 cat > /etc/sysctl.d/k8s.conf <<EOF
2 net.bridge.bridge-nf-call-ip6tables = 1
3 net.bridge.bridge-nf-call-iptables = 1
4 net.ipv4.ip_forward = 1
5 EOF
6 sysctl --system
```

加载IPVS模块

```
1 cat >/etc/sysconfig/modules/iptvs.modules<<EOF
2 #!/bin/bash
3 modprobe -- ip_vs
4 modprobe -- ip_vs_rr
5 modprobe -- ip_vs_wrr
6 modprobe -- ip_vs_sh
7 modprobe -- nf_conntrack_ipv4
8 EOF
9 chmod +x /etc/sysconfig/modules/iptvs.modules
10 source /etc/sysconfig/modules/iptvs.modules
11 lsmod | grep -e ip_vs -e nf_conntrack_ipv4
```

5.安裝配置Docker

配置阿里源

```
1 | cd /etc/yum.repos.d/
2 | wget https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo
3 | yum makecache fast
```

安装指定版本的Docker

```
1 | yum list docker-ce --showduplicates
2 | yum -y install docker-ce-19.03.15 docker-ce-cli-19.03.15
```

配置docker镜像加速和cgroup驱动:

```
1 | mkdir /etc/docker -p
2 | cat > /etc/docker/daemon.json <<EOF
3 | {
4 |     "registry-mirrors": ["https://ig21319y.mirror.aliyuncs.com"],
5 |     "exec-opts": ["native.cgroupdriver=systemd"]
6 | }
7 | EOF
```

启动Docker并设置开机自启动

```
1 | systemctl enable docker && systemctl start docker
```

检查版本

```
1 | docker -v
```

6.设置kubeadm的国内yum仓库

```
1 | cat >/etc/yum.repos.d/kubernetes.repo<<EOF
2 | [kubernetes]
3 | name=kubernetes
4 | baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-e17-x86_64/
5 | enabled=1
6 | gpgcheck=1
7 | repo_gpgcheck=1
8 | gpgkey=https://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg
   https://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg
9 | EOF
```

7.安装kubeadm

```
1 | yum list kubeadm --showduplicates
2 | yum install -y kubelet-1.19.3 kubeadm-1.19.3 kubectl-1.19.3 ipvsadm
```

8.设置kubelet开机启动

```
1 | systemctl enable kubelet
```

9.初始化集群命令-只在master节点运行

初始化命令：

```
1 | kubeadm init \
2 | --apiserver-advertise-address=10.0.0.10 \
3 | --image-repository registry.aliyuncs.com/google_containers \
4 | --kubernetes-version v1.19.3 \
5 | --service-cidr=10.1.0.0/16 \
6 | --pod-network-cidr=10.2.0.0/16 \
7 | --service-dns-domain=cluster.local \
8 | --ignore-preflight-errors=Swap \
9 | --ignore-preflight-errors=NumCPU
```

参数解释：

```
1 | --apiserver-advertise-address=10.0.0.11           #API Server的地址
2 | --image-repository registry.aliyuncs.com/google_containers   #镜像仓库地址,
这里使用的是阿里云
3 | --kubernetes-version v1.19.3                      #安装的k8s版本
4 | --service-cidr=10.1.0.0/16                         #Cluster IP
5 | --pod-network-cidr=10.2.0.0/16                     #Pod IP
6 | --service-dns-domain=cluster.local                #全域名的后缀，默认是
cluster.local
7 | --ignore-preflight-errors=Swap                   #忽略SWAP检查不通过的警告
8 | --ignore-preflight-errors=NumCPU                 #忽略CPU检查不通过的警告
```

这个步骤需要几分钟时间，执行完成后会有输出，这是node节点加入k8s集群的命令，需要自己保存到文本里

```
1 | kubeadm join 10.0.0.11:6443 --token fsteby.4pz9czptzvxhzrg2 \
2 |     --discovery-token-ca-cert-hash
sha256:8466da60f808479cf39c78fbbfc7c80b397ac95ed23164253abe05a9c8c619b4
```

10.为kubectl准备kubeconfig

```
1 | mkdir -p $HOME/.kube
2 | sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
3 | sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

11.获取node节点信息

```
1 | [root@node1 ~]# kubectl get nodes
2 | NAME      STATUS    ROLES      AGE      VERSION
3 | master    NotReady  master     15m      v1.19.3
```

12.设置kube-proxy使用ipvs模式

k8s默认使用的是iptables防火墙，可以修改为性能更高的ipvs模式

执行编辑命令，然后将mode: ""修改为mode: "ipvs"然后保存退出，这里涉及到configmap资源类型，后面详细讲解。

```
1 | kubectl edit cm kube-proxy -n kube-system
```

重启kube-proxy

```
1 | kubectl -n kube-system get pod|grep kube-proxy|awk '{print "kubectl -n kube-system delete pod \"$1\""}'|bash
```

查看pod信息

```
1 | kubectl get -n kube-system pod|grep "kube-proxy"
```

检查日志，如果出现IPVS rr就表示成功

```
1 | [root@node1 ~]# kubectl -n kube-system logs -f kube-proxy-7cdbn
2 | I0225 08:03:57.736191      1 node.go:135] Successfully retrieved node IP:
3 |          10.0.0.11
4 | I0225 08:03:57.736249      1 server_others.go:176] Using ipvs Proxier.
5 | W0225 08:03:57.736841      1 proxier.go:420] IPVS scheduler not specified,
6 |          use rr by default
```

检查IPVS规则

```
1 | [root@node1 ~]# ipvsadm -Ln
2 | IP Virtual Server version 1.2.1 (size=4096)
3 | Prot LocalAddress:Port Scheduler Flags
4 |     -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
5 | TCP  10.1.0.1:443 rr
6 |     -> 10.0.0.11:6443            Masq    1      0          0
7 | TCP  10.1.0.10:53 rr
8 | TCP  10.1.0.10:9153 rr
9 | UDP  10.1.0.10:53 rr
```

13. 警告解决

告警内容：

```
1 | [init] Using Kubernetes version: v1.20.4
2 | [preflight] Running pre-flight checks
3 |         [WARNING NumCPU]: the number of available CPUs 1 is less than the
4 |         required 2
5 |         [WARNING IsDockerSystemdCheck]: detected "cgroupfs" as the Docker
6 |         cgroup driver. The recommended driver is "systemd". Please follow the guide at
7 |         https://kubernetes.io/docs/setup/cri/
8 |         [WARNING SystemVerification]: this Docker version is not on the list
9 |         of validated versions: 20.10.7. Latest validated version: 19.03
```

上面的告警信息主要有几点

1. kubeadm和docker的版本不对，提示docker的版本太高
2. Docker的cgroup driver驱动模式不对，建议使用systemd

14. 部署Flannel网络插件-只在master节点上安装部署

克隆代码

```
1 | git clone --depth 1 https://github.com/coreos/flannel.git
```

下载下来后还需要修改资源配置清单

```
1 #修改配置文件，将128行替换为PodIP，在189行新增加一行指定网卡名
2 cd flannel/Documentation/
3 vim kube-flannel.yml
4 128:      "Network": "10.2.0.0/16",
5 189:          - --iface=eth0
```

应用资源配置清单

```
1 | kubectl create -f kube-flannel.yml
```

检查pod运行状态，等一会应该全是running

```
1 [root@node1 ~]# kubectl -n kube-system get pod
2 NAME                               READY   STATUS    RESTARTS   AGE
3 coredns-6d56c8448f-2w8gz           1/1     Running   0          12m
4 coredns-6d56c8448f-6k7wf           1/1     Running   0          12m
5 etcd-node1                          1/1     Running   0          12m
6 kube-apiserver-node1              1/1     Running   0          12m
7 kube-controller-manager-node1     1/1     Running   0          12m
8 kube-flannel-ds-2nmgt             1/1     Running   0          99s
9 kube-proxy-nmk2b                  1/1     Running   0          12m
10 kube-scheduler-node1             1/1     Running   0          12m
```

15.node节点执行加入集群命令-所有的Node节点执行

刚才在master集群初始化的时候会生成一串命令，这个命令就是给node节点加入集群的指令。

```
1 | kubeadm join 10.0.0.11:6443 --token uqf018.mia8v3i1zcai19sj --discovery-
  token-ca-cert-hash
  sha256:e7d36e1fb53e59b12f0193f4733edb465d924321bcfc055f801cf1ea59d90aae
```

如果当时没有保存，可以执行以下命令重新查看：

```
1 | kubeadm token create --print-join-command
```

16.查看集群状态

在master节点上查看节点状态

```
1 | kubectl get nodes
```

17.给节点打标签

在master节点上设置节点标签

```
1 | kubectl label nodes node1 node-role.kubernetes.io/node=
2 | kubectl label nodes node2 node-role.kubernetes.io/node=
```

18.再次查看节点状态

全部都是Ready就是正常状态

```
1 [root@node1 Documentation]# kubectl get nodes
2 NAME     STATUS   ROLES    AGE      VERSION
3 node1    Ready    master   7m18s   v1.19.3
4 node2    Ready    node     6m40s   v1.19.3
5 node3    Ready    node     6m50s   v1.19.3
```

19.支持命令补全

```
1 yum install bash-completion -y
2 source /usr/share/bash-completion/bash_completion
3 source <(kubectl completion bash)
4 kubectl completion bash >/etc/bash_completion.d/kubectl
```

第6章 常用资源类型

1.工作负载类型

```
1 RC  ReplicaController
2 RS  ReplicaSet
3 DP  Deployment
4 DS  DaemonSet
```

2.服务发现及负载均衡

```
1 Service
2 Ingress
```

3.配置与存储资源

```
1 ConfigMap 存储配置文件
2 Secret     存储用户字典
```

4.集群级别资源

```
1 Namespace
2 Node
3 Role
4 ClusterRole
5 RoleBinding
6 ClusterRoleBinding
```

第7章 资源配置清单体验

1.创建资源的方法

- 1 在Docker时代，我们可以使用docker run或者docker-compose来管理容器。但是在k8s里，我们使用json格式或者YAML格式的资源清单来描述应用。
- 2 但是相对来说，yaml格式更容易理解和阅读，所以我们都是使用yaml格式来声明。

2. 资源清单介绍

查看资源清单所需字段

```
1 kubectl explain pod  
2 kubectl explain pod  
3 kubectl explain pod.spec  
4 kubectl explain pod.spec.volumes
```

资源清单字段介绍

```
1 apiVersion: v1 #属于k8s哪一个API版本或组  
2 kind: Pod #资源类型  
3 metadata: #元数据，嵌套字段  
4 spec: #定义容器的规范，创建的容器应该有哪些特性  
5 status: #只读的，由系统控制，显示当前容器的状态
```

3. 使用资源配置清单创建POD

3.1 首先使用命令行创建一个pod

```
1 kubectl create deployment nginx --image=nginx:alpine  
2 kubectl get pod -o wide
```

3.2 将刚才创建的pod配置到处成yaml格式

```
1 kubectl get pod -o yaml > nginx-pod.yaml
```

3.3 精简资源清单，删掉不需要的配置

```
1 apiVersion: v1  
2 kind: Pod  
3 metadata:  
4   name: nginx  
5   labels:  
6     app: nginx  
7 spec:  
8   containers:  
9     - name: nginx  
10       image: nginx:alpine  
11       imagePullPolicy: IfNotPresent  
12     ports:  
13       - name: http  
14         containerPort: 80
```

json格式写法：

```
1 |
2 |     apiVersion: "v1",
3 |     kind: "Pod",
4 |     metadata:
5 |         {
6 |             name: "nginx",
7 |             labels:
8 |                 {
9 |                     app: "nginx"
10 |                 }
11 |         }
12 |     spec:
13 |         {
14 |             containers:
15 |                 {
16 |                     name: "nginx",
17 |                     image: "nginx:alpine",
18 |                     imagePullPolicy: "IfNotPresent"
19 |                 }
20 |         }
21 | }
```

3.5 删除命令行创建的资源

```
1 | kubectl delete deployments.apps nginx
```

3.6 应用资源配置清单

```
1 | kubectl create -f nginx-pod.yaml
```

3.7 查看pod信息

```
1 | kubectl get pod -o wide
```

3.8 查看pod详细信息

```
1 | kubectl describe pod nginx
```

4.POD资源清单总结

声明式管理 我想运行一个Nginx k8s帮你干活

```
1 apiVersion: v1      #api版本号
2 kind: Pod          #资源类型
3 metadata:          #原数据
4   name: nginx       #pod显示名称
5   labels:           #标签
6     app: nginx      #具体的标签
7 spec:              #定义pod参数
8   containers:       #定义容器
9     - image: nginx:alpine    #镜像名称
10    imagePullPolicy: IfNotPresent  #拉取镜像策略, 如果本地已经存在镜像, 就不重新拉
11      取
11     name: nginx      #容器名称
```

第8章 标签的使用设置

1.Node标签的设置使用

1.1 查看node的标签

```
1 | kubectl get node --show-labels
```

1.2 给node打标签

```
1 | kubectl label nodes node1 CPU=Xeon
2 | kubectl label nodes node2 disktype=ssd
```

1.3 编辑POD资源配置清单, 使用node标签选择器

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: nginx
5   labels:
6     app: nginx
7 spec:
8   containers:
9     - name: nginx
10    image: nginx:1.14.0
11    imagePullPolicy: IfNotPresent
12    ports:
13      - name: http
14        containerPort: 80
15    nodeSelector:
16      #CPU: Xeon
17      disktype: SSD
```

1.4 删除容器重新创建

```
1 | kubectl delete pod nginx
2 | kubectl create -f nginx-pod.yaml
```

1.5 查看结果

```
1 | kubectl get pod -o wide
```

1.6 删除节点标签

```
1 | kubectl label nodes node2 CPU-
2 | kubectl label nodes node3 disktype-
```

2.Pod标签设置

2.1 标签说明

一个标签可以给多个POD使用
一个POD也可以拥有多个标签

2.2 查看POD标签

```
1 | kubectl get pod --show-labels
```

2.3 添加标签方法

方法1:直接编辑资源配置清单:

```
1 | apiVersion: v1
2 | kind: Pod
3 | metadata:
4 |   name: nginx
5 |   labels:
6 |     app: nginx
7 |     release: beta
```

方法2:命令行打标签

```
1 | kubectl label pods nginx release=beta
2 | kubectl label pods nginx job=linux
3 | kubectl get pod --show-labels
```

2.4 删除标签

```
1 | kubectl label pod nginx job-
2 | kubectl get pod --show-labels
```

2.5 POD标签实验

生成2个不同标签的POD

```
1 | kubectl create deployment nginx --image=nginx:1.14.0
2 | kubectl get pod --show-labels
3 | kubectl label pods nginx-xxxxxxx release=stable
4 | kubectl get pod --show-labels
```

根据标签查看

```
1 | kubectl get pods -l release=beta --show-labels
2 | kubectl get pods -l release=stable --show-labels
```

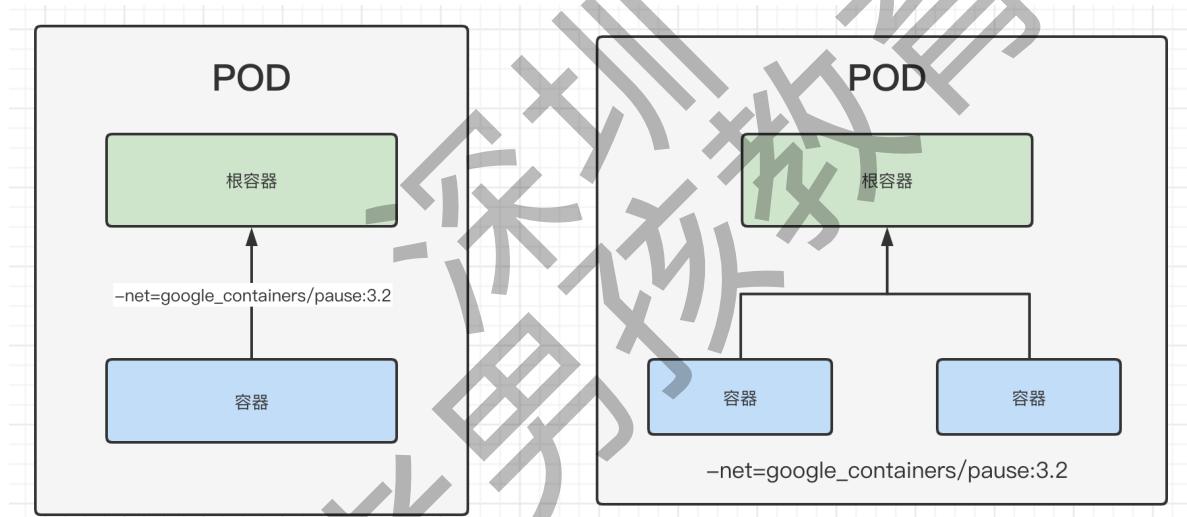
根据标签删除

```
1 | kubectl delete pod -l app=nginx
```

第9章 重新认识POD

1.Pod的原理

1.1 共享网络空间

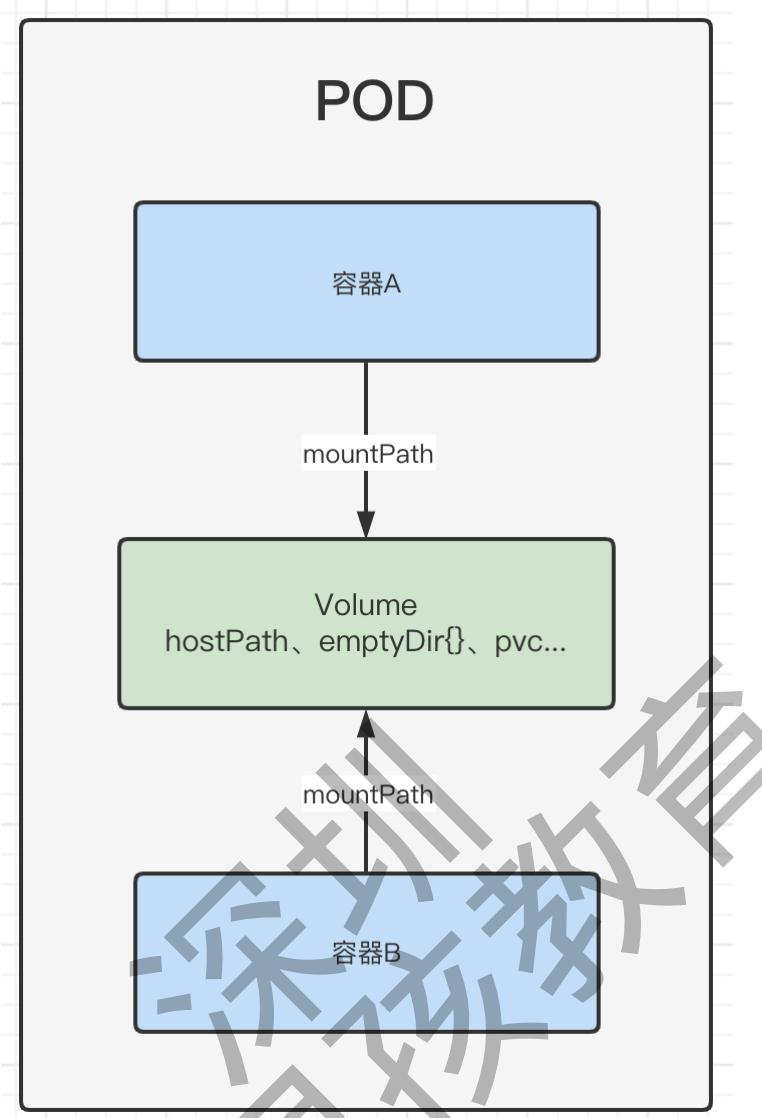


- 1 POD内的容器使用container模式共享根容器的网络
- 2 容器看到的网络设备信息和根容器完全相同
- 3 POD内的多个容器可以直接使用localhost通信
- 4 POD内的多个容器不能绑定相同的端口
- 5 POD的生命周期和根容器一样，如果根容器推出，POD就退出了

可以在node节点使用docker inspect查看容器的详细信息，会发现有网络模式为container

```
1 | [root@node2 ~]# docker inspect 6d636151f567|grep "NetworkMode"
2 |           "NetworkMode":
3 |             "container:1613f2dee9e410805679c6073ee7c1a9160c3fd24b7135f7e5ed17ffc6502ee2",
```

1.2 共享文件系统



- 1 默认情况下一个POD内的容器文件系统是互相隔离的
- 2 如果想让一个POD的容器共享文件，那么只需要定义一个volume，然后两个容器分别挂在这个volume就可以共享了

举个例子：

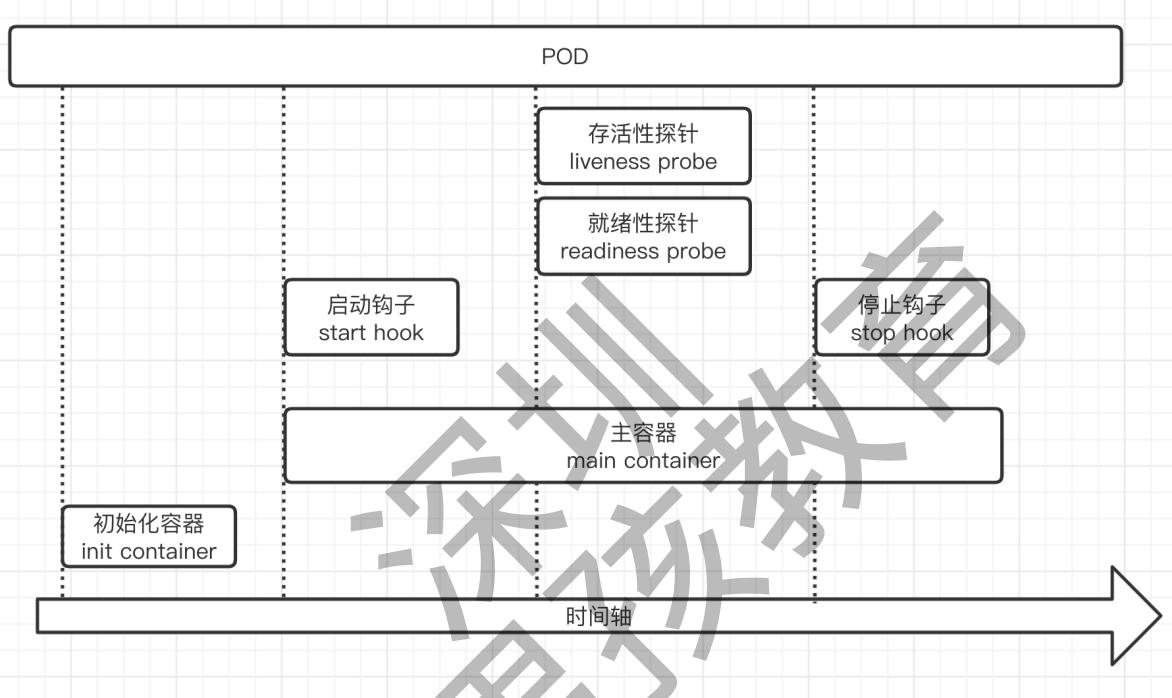
```
1 cat > nginx_vo.yaml << 'EOF'
2 apiVersion: v1
3 kind: Pod
4 metadata:
5   name: nginx
6 spec:
7   nodeName: node2
8
9   volumes:
10  - name: nginxlog
11    hostPath:
12      path: /var/log/nginx/
13
14   containers:
15   - name: nginx-1
16     image: nginx
```

```

17   volumeMounts:
18     - name: nginxlog
19       mountPath: /var/log/nginx/
20
21     - name: tail-log
22       image: busybox
23       args: [/bin/sh, -c, 'tail -f /var/log/nginx/access.log']
24       volumeMounts:
25         - name: nginxlog
26           mountPath: /var/log/nginx/
27 EOF

```

2.Pod的生命周期



```

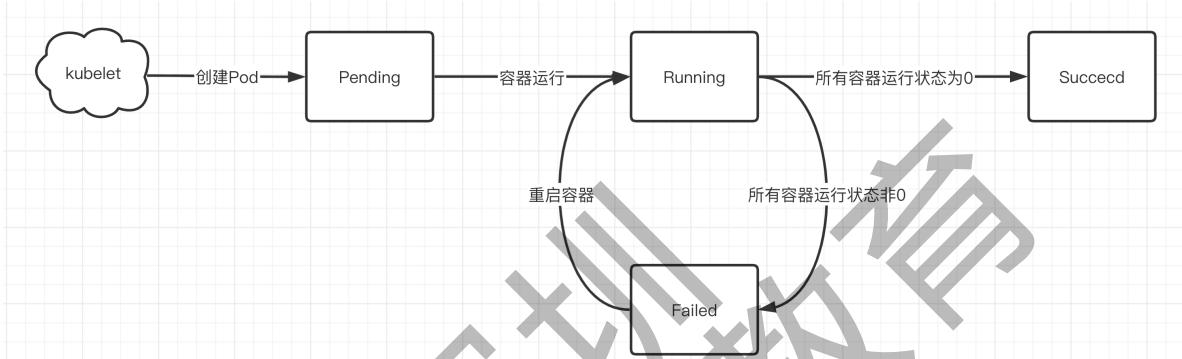
1 init container:
2 初始化容器是指在主容器启动之前，先启动一个容器来做一些准备工作，比如两个容器做了共享
volum, 然后可以先启动一个容器来对目录进行更改用户授权。
3 比如主容器需要连接数据库，可以先使用初始化容器测试可否正常连接数据库，如果可以正常连接再启
动主容器。
4
5 hook:
6 PostStart: 在容器启动创建后立刻执行，但是时间不能太长，否则容器将不会是running状态
7 PreStop: 在容器停止被删除前执行，主要用于优雅的关闭应用程序。
8
9 liveness probe:
10 存活性探针，用于确定容器内的应用是否还活着
11
12 readiness probe:
13 就绪性探针，用于确定容器是否已经准备就绪可以干活了，比如扩容一个Pod，只有等这个Pod里面的
应用完全启动了，才会将流量进入。

```

3.POD运行状态

1 Pending (挂起) :

2 Pod 已被Kubernetes 系统接受，但有一个或者多个容器尚未创建亦未运行。此阶段包括等待 Pod 被调度的时间和通过网络下载镜像的时间
3
4 **Running (运行中) :**
5 Pod 已经绑定到了某个节点，Pod 中所有的容器都已被创建。至少有一个容器仍在运行，或者正处于启动或重启状态。
6
7 **Succeeded(成功) :**
8 Pod 中的所有容器都已成功终止，并且不会再重启。
9
10 **Failed (失败) :**
11 Pod 中的所有容器都已终止，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。
12
13 **Unknown (未知) :**
14 因为某些原因无法取得 Pod 的状态。这种情况通常是因为与 Pod 所在主机通信失败。



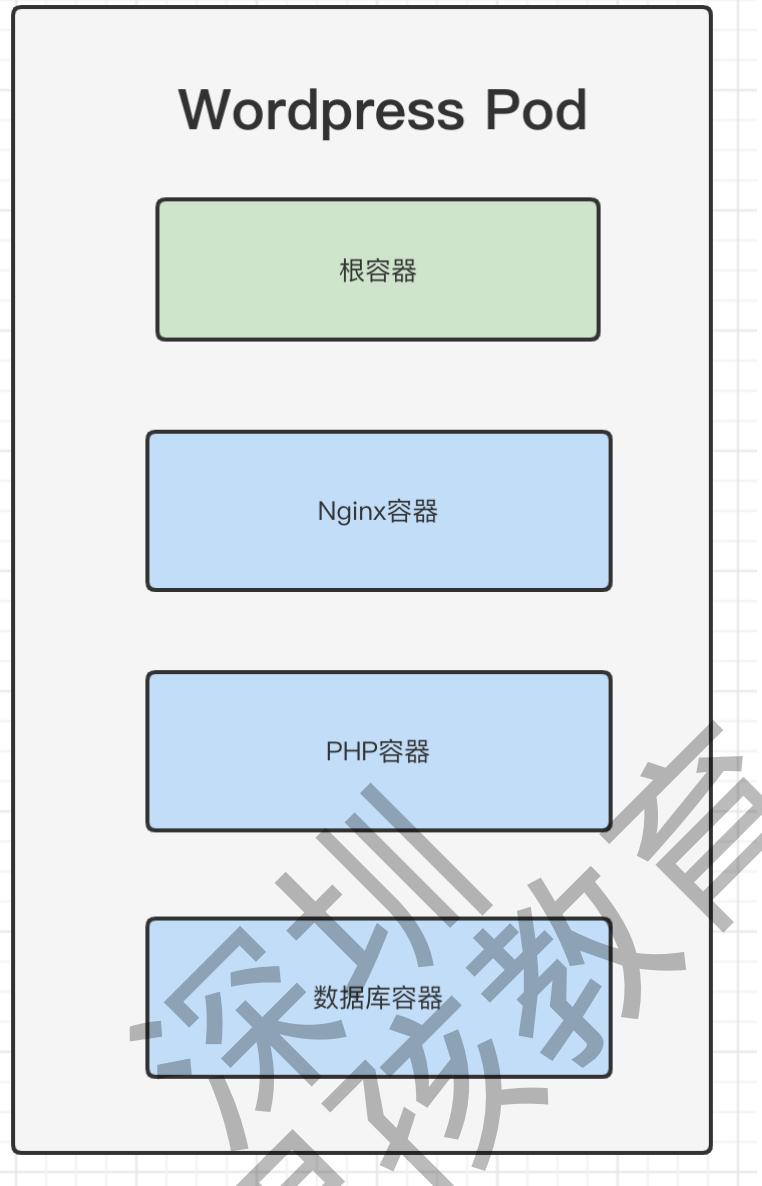
4.Pod对容器的封装和应用

那么在工作中我们究竟如何决策一个POD是跑一个容器还是多个容器呢？

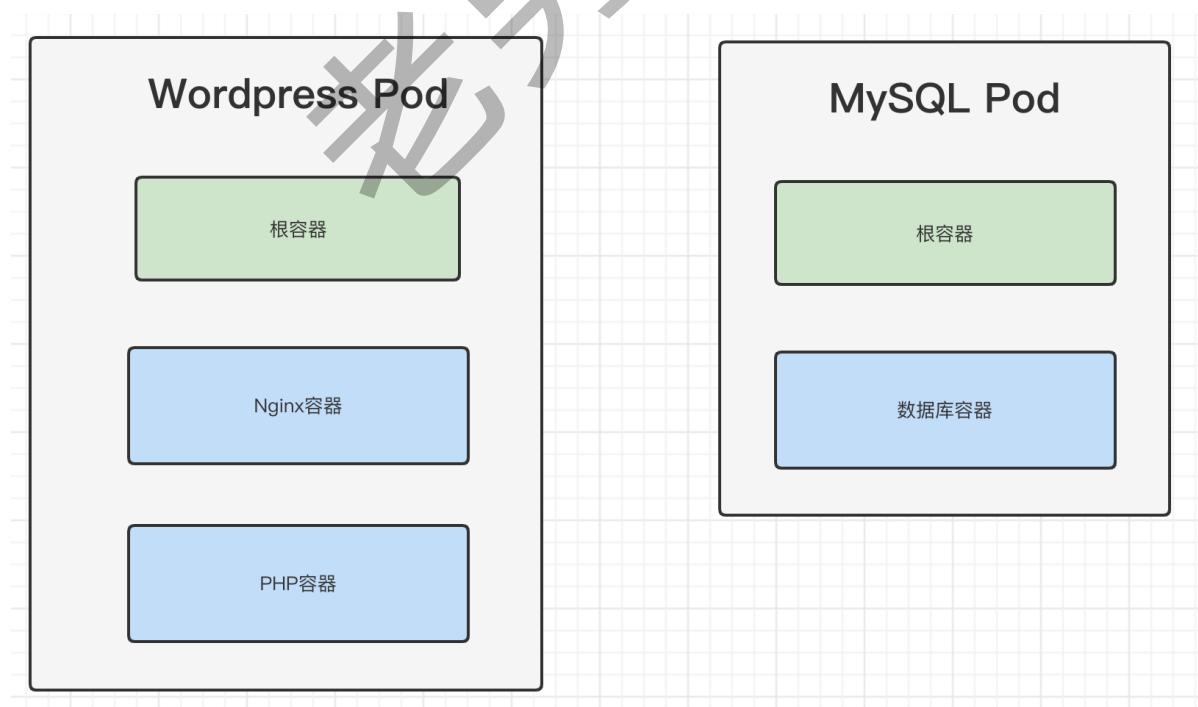
在实际工作中我们除了完成任务以外还需要考虑以后扩容的问题，就拿wordpress举例，有以下两种方案：

第一种：全部放一个pod里

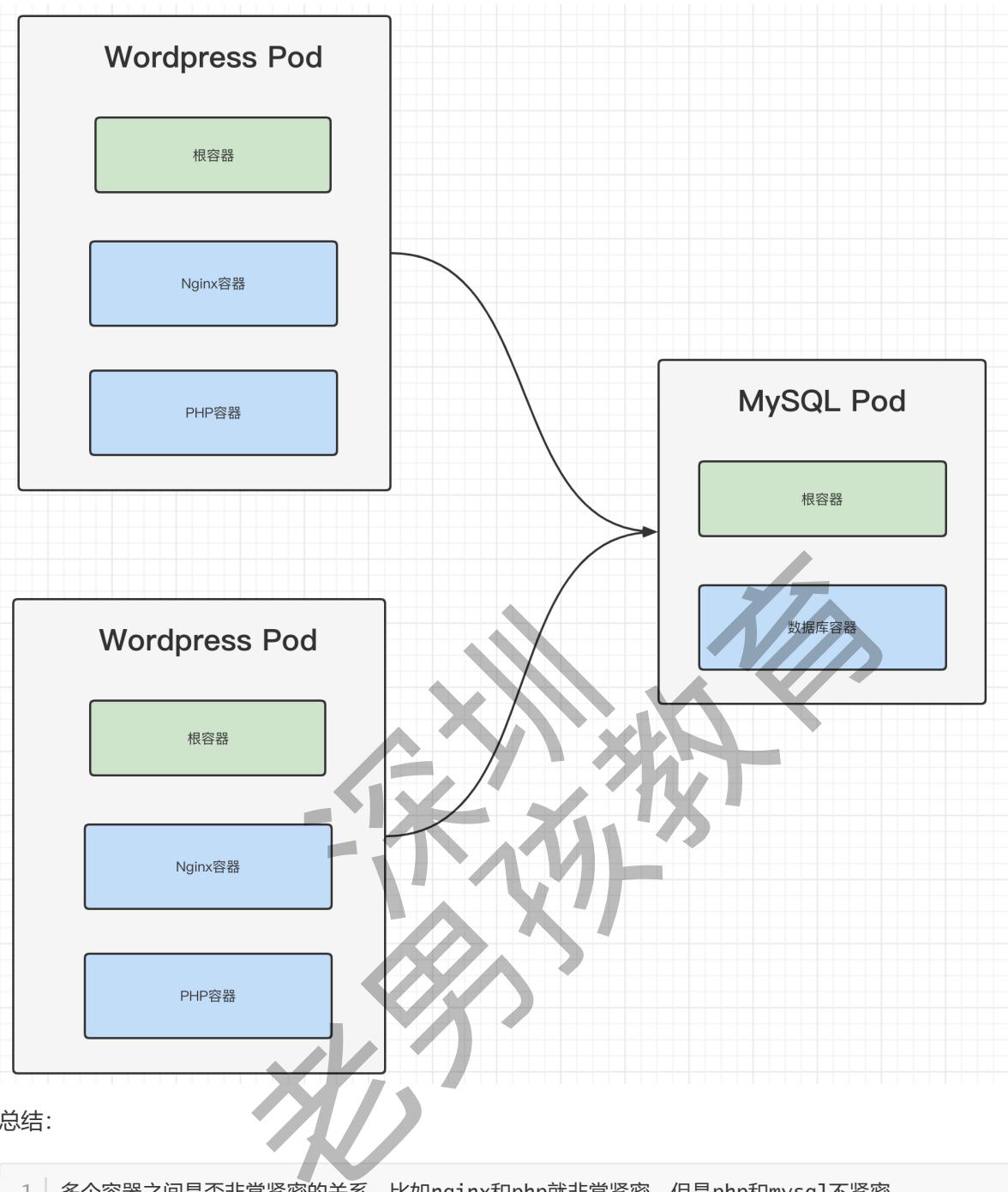
Wordpress Pod



第二种：Wordpress和MySQL分开



那么如何扩容呢？如果第一种方案大家会发现没有办法很好的扩容，因为数据库和wordpress已经绑定成一个整体了，扩容wordpress就得扩容mysql。而第二种方案就要灵活的多。



总结：

- 1 | 多个容器之间是否非常紧密的关系，比如nginx和php就非常紧密，但是php和mysql不紧密
- 2 | 这些容器是否必须是一个整体
- 3 | 这些容器放在一起是否影响扩容

5. 初始化容器

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx-init
5  spec:
6    nodeName: node2
7
8    volumes:
9      - name: nginx-index
10        emptyDir: {}
```

```
11 initContainers:
12   - name: init
13     image: busybox
14     args: [/bin/sh, -c, 'echo k8s >> /usr/share/nginx/html/index.html']
15     volumeMounts:
16       - name: nginx-index
17         mountPath: "/usr/share/nginx/html"
18
19 containers:
20   - name: nginx
21     image: nginx:alpine
22     ports:
23       - containerPort: 80
24     volumeMounts:
25       - name: nginx-index
26         mountPath: "/usr/share/nginx/html"
```

5.Pod hook

PostStart:

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx-hook
5  spec:
6    nodeName: node2
7
8    containers:
9      - name: nginx
10        image: nginx:alpine
11        lifecycle:
12          postStart:
13            exec:
14              command: [/bin/sh, -c, 'echo k8s >
15                /usr/share/nginx/html/index.html']
16            ports:
17              - containerPort: 80
```

PostStop:

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx-hook
5  spec:
6    nodeName: node2
7
8    volumes:
9      - name: nginxlog
10        hostPath:
11          path: /var/log/nginx/
12
13   containers:
14     - name: nginx
```

```
15   image: nginx:alpine
16   lifecycle:
17     postStart:
18       exec:
19         command: [/bin/sh, -c, 'echo k8s >
/usr/share/nginx/html/index.html']
20     preStop:
21       exec:
22         command: [/bin/sh, -c, 'echo bye > /var/log/nginx/stop.log']
23   ports:
24     - containerPort: 80
25
26   volumeMounts:
27     - name: nginxlog
28       mountPath: /var/log/nginx/
```

6.Pod 健康检查

6.1 存活探针

1 | 存活探针简单来说就是用来检测容器的应用程序是否还正常工作，如果应用程序不正常，即使容器还活着也没有意义了，所以这时候就可以使用存活探针来探测，如果应用程序不正常，就重启POD。

存活探针语法：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: liveness-pod
5  spec:
6    nodeName: node2
7
8    volumes:
9      - name: nginx-html
10        hostPath:
11          path: /usr/share/nginx/html/
12
13   containers:
14     - name: liveness
15       image: nginx
16       imagePullPolicy: IfNotPresent
17
18   lifecycle:
19     postStart:
20       exec:
21         command: [/bin/sh, -c, 'echo k8s >
/usr/share/nginx/html/index.html']
22
23   livenessProbe:
24     exec:
25       command:
26         - cat
27         - /usr/share/nginx/html/index.html
28     initialDelaySeconds: 3
29     periodSeconds: 1
30
```

```
31   volumeMounts:  
32     - name: nginx-html  
33       mountPath: /usr/share/nginx/html/
```

配置解释：

```
1   livenessProbe: #存活探针  
2     exec: #执行命令  
3       command: #具体的命令结果0的状态被视为存活,  
非零是不健康的。  
4         - cat  
5         - /tmp/healthy  
6     initialDelaySeconds: 3 #第一次执行探针的时候等待5秒  
7     periodSeconds: 1 #每隔5秒执行一次存活探针，默认为10秒，最小值  
为1秒
```

通过命令可以查看Pod的详细状态

```
1 | kubectl describe pod liveness-pod  
2 | kubectl get pod -w
```

除了我们自己写的命令以外，也支持基于http请求：

```
1 | apiVersion: v1  
2 | kind: Pod  
3 | metadata:  
4 |   name: liveness-pod  
5 | spec:  
6 |   nodeName: node2  
7 |  
8 |   volumes:  
9 |     - name: nginx-html  
10 |       hostPath:  
11 |         path: /usr/share/nginx/html/  
12 |  
13 |   containers:  
14 |     - name: liveness  
15 |       image: nginx  
16 |       imagePullPolicy: IfNotPresent  
17 |  
18 |   lifecycle:  
19 |     postStart:  
20 |       exec:  
21 |         command: [/bin/sh, -c, 'echo k8s >  
/usr/share/nginx/html/index.html']  
22 |  
23 |   livenessProbe:  
24 |     #exec:  
25 |     #   command:  
26 |     #   - cat  
27 |     #   - /usr/share/nginx/html/index.html  
28 |     httpGet:  
29 |       path: /index.html  
30 |       port: 80  
31 |     initialDelaySeconds: 3  
32 |     periodSeconds: 1
```

```
33  
34     volumeMounts:  
35       - name: nginx-html  
36         mountPath: /usr/share/nginx/html/
```

参数解释：

```
1   livenessProbe:  
2     httpGet: #基于http请求探测  
3       path: /health.html #请求地址, 如果这个地址返回的状态码在200-400之间  
正常  
4       port: 80 #请求的端口  
5     initialDelaySeconds: 3 #第一次启动探测在容器启动后3秒开始  
6     periodSeconds: 3 #容器启动后每隔3秒检查一次
```

6.3 就绪探针

有时候我们Pod本身已经起来了，但是pod的容器还没有完全准备好对外提供服务，那么这时候流量进来就会造成请求失败的情况出现，针对这种情况k8s有一种探针叫就绪探针，他的作用就是让k8s知道你的Pod内应用是否准备好为请求提供服务。只有就绪探针ok了才会把流量转发到pod上。

配置语法：就绪探针的配置语法和存活探针基本一样

```
1  apiVersion: v1  
2  kind: Pod  
3  metadata:  
4    name: liveness-pod  
5  spec:  
6    containers:  
7      - name: liveness-pod  
8        image: nginx  
9        lifecycle:  
10          postStart:  
11            exec:  
12              command: ["/bin/sh", "-c", "echo ok > /usr/share/nginx/html/health.html"]  
13          readinessProbe:  
14            httpGet:  
15              path: /actuator/health  
16              port: 8080  
17              initialDelaySeconds: 5  
18              timeoutSeconds: 3  
19              periodSeconds: 3  
20              successThreshold: 3  
21              failureThreshold: 3  
22          livenessProbe:  
23            httpGet:  
24              path: /health.html  
25              port: 80  
26              initialDelaySeconds: 3  
27              periodSeconds: 3
```

参数解释：

1	initialDelaySeconds:	第一次执行探针需要在容器启动后等待的时候时间
2	periodSeconds:	容器启动后每隔多少秒执行一次存活探针
3	timeoutSeconds:	探针超时时间，默认1秒，最小1秒
4	successThreshold:	探针失败后最少连续探测成功多少次才被认定成功，默认1次，如果是liveness必须为1
5	failureThreshold:	探针成功后被视为失败的探测的最小连续失败次数。默认3次。最小值为1

7.Pod资源限制

- 1 在学习Docker的时候我们知道容器可以使用Linux系统的CGroup技术限制内存和CPU的使用率，那么POD里应该如何限制容器的CPU和内存资源呢？

资源配置清单：

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: resource-demo
5 spec:
6   containers:
7     - name: resource-demo
8       image: nginx
9       ports:
10      - containerPort: 80
11    resources:
12      requests:
13        memory: 50Mi
14        cpu: 1500m
15      limits:
16        memory: 100Mi
17        cpu: 200m
```

参数解释：

- 1 **requests** :节点所需的最小计算资源量，k8s调度的时候的依据值
- 2 **limits** :限制允许的最大计算资源量，真正的资源限制参数

数值的转换：

```
1 1 CPU = 1000m
2 0.5 CPU = 500m
3 1 Mib = 1024 kib
4 1 MB = 1000 KB
```

查看验证：

```
1 docker inspect 容器ID|grep CgroupParent
2 cd /sys/fs/cgroup/cpu/kubepods.slice/kubepods-burstable.slice/kubepods-
burstable-podxxxx.slice
3 cat cpu.cfs_quota_us
```

第10章 Pod控制器

1. 控制器作用

1. pod类型的资源，删除pod后，不会重建
2. 替用户监视并保证相应的节点上始终有用户所期望的副本数量的pod在运行
3. 如果所运行的pod副本数超过了用户期望的，那么控制器就会删掉，直到和用户期望的一致
4. 如果所运行的pod副本数低于用户期望的，那么控制器就会创建，直到和用户期望的一致

2. 常用控制器类型

```
1 ReplicaSet RS:  
2 按用户期望的副本创建pod，并始终保持相应数量副本  
3  
4 Deployment:  
5 Deployment通过控制RS来保证POD始终保持相应的数量副本  
6 支持滚动更新，回滚，回滚默认保留10个版本  
7 提供声明式配置，支持动态修改  
8 管理无状态应用最理想的控制器  
9 node节点可能会运行0个或多个POD  
10  
11 DaemonSet:  
12 一个节点只运行一个，必须是始终运行的状态  
13  
14 StatefulSet:  
15 有状态应用
```

4. ReplicaSet控制器

<https://kubernetes.io/zh/docs/concepts/workloads/controllers/relicaset/>

4.1 编写RS控制器资源配置清单

```
1 apiVersion: apps/v1 #接口版本号  
2 kind: Replicaset #资源类型 Replicaset  
3 metadata:  
4   name: nginx-rs #RS的原数据  
5   labels:  
6     app: nginx-rs #RS原数据名称  
7 spec:  
8   replicas: 2 #RS具体标签  
9   selector:  
10    matchLabels:  
11      app: nginx-pod #定义pod的实际运行配置  
12    template:  
13      metadata:  
14        labels:  
15          app: nginx-pod #要运行几个Pod  
16        spec:  
17          containers:  
18            - name: nginx #选择器  
19              image: nginx:1.14 #匹配标签  
20              imagePullPolicy: IfNotPresent #匹配Pod的标签  
21            ports:
```

#pod自己的原数据
#pod自己的标签
#pod具体标签名
#定义容器运行的配置
#容器参数
#容器名
#容器镜像
#镜像拉取策略
#暴露端口

```
22     - name: http          #端口说明  
23       containerPort: 80    #容器暴露的端口
```

4.2.应用RS资源配置清单

```
1 | kubectl create -f nginx-rs.yaml
```

4.3.查看RS资源

```
1 | kubectl get rs  
2 | kubectl get pod -o wide
```

4.4.修改yaml文件应用修改

```
1 | vim nginx-rs.yaml  
2 | kubectl apply -f nginx-rs.yaml
```

4.5.动态修改配置 扩容 收缩 升级

```
1 | kubectl edit rs nginx  
2 | kubectl scale rs nginx --replicas=5
```

5.Deployment控制器

5.1 Deployment和RS控制器的关系

- 1 虽然我们创建的是Deployment类型资源，但实际上控制副本还是由RS来控制的，Deployment只是替我们去创建RS控制器，然后再由RS去控制POD副本。
- 2
- 3 Deployment控制器还有一个比较重要的功能就是滚动更新和版本回滚，而这个功能就是以来RS控制器。

5.2 资源配置清单

```
1 | cat >nginx-dp.yaml <<EOF  
2 | apiVersion: apps/v1  
3 | kind: Deployment  
4 | metadata:  
5 |   name: nginx-deployment  
6 |   namespace: default  
7 | spec:  
8 |   replicas: 2  
9 |   selector:  
10 |     matchLabels:  
11 |       app: nginx  
12 |     template:  
13 |       metadata:  
14 |         name: nginx-pod  
15 |         labels:  
16 |           app: nginx  
17 |       spec:
```

```
18     containers:  
19         - name: nginx-containers  
20             image: nginx:1.14.0  
21             imagePullPolicy: IfNotPresent  
22             ports:  
23                 - name: http  
24                     containerPort: 80  
25 EOF
```

5.2.应用资源配置清单

```
1 | kubectl create -f nginx-dp.yaml
```

5.3.查看

```
1 | kubectl get pod -o wide  
2 | kubectl get deployments.apps  
3 | kubectl describe deployments.apps nginx-deployment
```

5.4.更新版本

方法1: 命令行根据资源配置清单修改镜像

```
1 | kubectl set image -f nginx-dp.yaml nginx-containers=nginx:1.16.0
```

查看有没有更新

```
1 | kubectl get pod  
2 | kubectl describe deployments.apps nginx-deployment  
3 | kubectl describe pod nginx-deployment-7c596b4d95-6ztld
```

方法2: 命令行根据资源类型修改镜像

打开2个窗口:

第一个窗口监控pod状态

```
1 | kubectl get pod -w
```

第二个窗口更新操作

```
1 | kubectl set image deployment nginx-deployment nginx-containers=nginx:1.14.0
```

查看更新后的deployment信息

```
1 | kubectl describe deployments.apps nginx-deployment
2 | -----
3 |     Normal  ScalingReplicaSet  14m           deployment-controller
4 |       Scaled up replica set nginx-deployment-7c596b4d95 to 1
5 |     Normal  ScalingReplicaSet  14m           deployment-controller
6 |       Scaled down replica set nginx-deployment-9c74bb6c7 to 1
7 |     Normal  ScalingReplicaSet  14m           deployment-controller
8 |       Scaled up replica set nginx-deployment-7c596b4d95 to 2
9 |     Normal  ScalingReplicaSet  13m           deployment-controller
10 |      Scaled down replica set nginx-deployment-9c74bb6c7 to 0
11 |    Normal  ScalingReplicaSet  8m30s         deployment-controller
12 |      Scaled up replica set nginx-deployment-9c74bb6c7 to 1
13 |    Normal  ScalingReplicaSet  8m29s (x2 over 32m) deployment-controller
14 |      Scaled up replica set nginx-deployment-9c74bb6c7 to 2
15 |    Normal  ScalingReplicaSet  8m29s         deployment-controller
16 |      Scaled down replica set nginx-deployment-7c596b4d95 to 1
17 |    Normal  ScalingReplicaSet  8m28s         deployment-controller
18 |      Scaled down replica set nginx-deployment-7c596b4d95 to 0
```

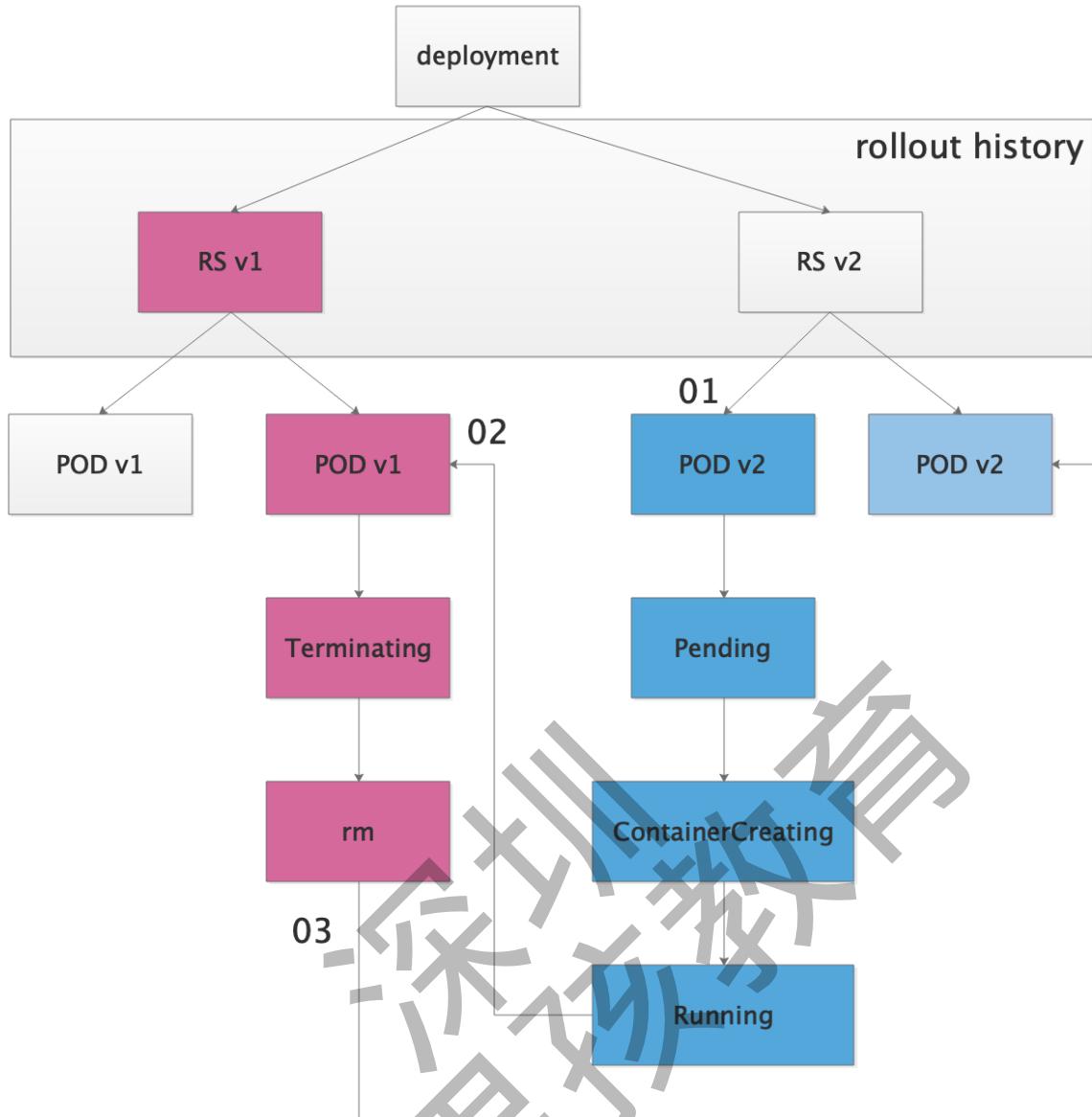
更新过程：

```
1 | nginx-deployment-7c596b4d95-8z7kf #老的版本
2 | nginx-deployment-7c596b4d95-6ztld #老的版本
3 |
4 | nginx-deployment-9c74bb6c7-pgfixz 0/1 Pending
5 | nginx-deployment-9c74bb6c7-pgfixz 0/1 Pending
6 | nginx-deployment-9c74bb6c7-pgfixz 0/1 ContainerCreating #拉取新版本镜像
7 | nginx-deployment-9c74bb6c7-pgfixz 1/1 Running #运行新POD
8 | nginx-deployment-7c596b4d95-8z7kf 1/1 Terminating #停止一个旧的
9 |   POD
10 | nginx-deployment-9c74bb6c7-h7mk2 0/1 Pending
11 | nginx-deployment-9c74bb6c7-h7mk2 0/1 Pending
12 | nginx-deployment-9c74bb6c7-h7mk2 0/1 ContainerCreating #拉取新版本镜像
13 | nginx-deployment-9c74bb6c7-h7mk2 1/1 Running #运行新POD
14 | nginx-deployment-7c596b4d95-6ztld 1/1 Terminating #停止一个旧的
15 |   POD
16 | nginx-deployment-7c596b4d95-8z7kf 0/1 Terminating #等待旧的POD结
17 |   束
18 | nginx-deployment-7c596b4d95-6ztld 0/1 Terminating #等待旧的POD结
19 |   束
```

查看滚动更新状态：

```
1 | kubectl rollout status deployment nginx-deployment
```

滚动更新示意图：



5.5. 回滚上一个版本

```
1 | kubectl describe deployments.apps nginx-deployment
2 | kubectl rollout undo deployment nginx-deployment
3 | kubectl describe deployments.apps nginx-deployment
```

5.6. 回滚到指定版本

v1 1.14.0

v2 1.15.0

v3 3.333.3

回滚到v1版本

创建第一版 1.14.0

```
1 | kubectl create -f nginx-dp.yaml --record
```

更新第二版 1.15.0

```
1 | kubectl set image deployment nginx-deployment nginx-containers=nginx:1.15.0
```

更新第三版 1.99.0

```
1 | kubectl set image deployment nginx-deployment nginx-containers=nginx:1.16.0
```

查看所有历史版本

```
1 | kubectl rollout history deployment nginx-deployment
```

查看指定历史版本信息

```
1 | kubectl rollout history deployment nginx-deployment --revision=1
```

回滚到指定版本

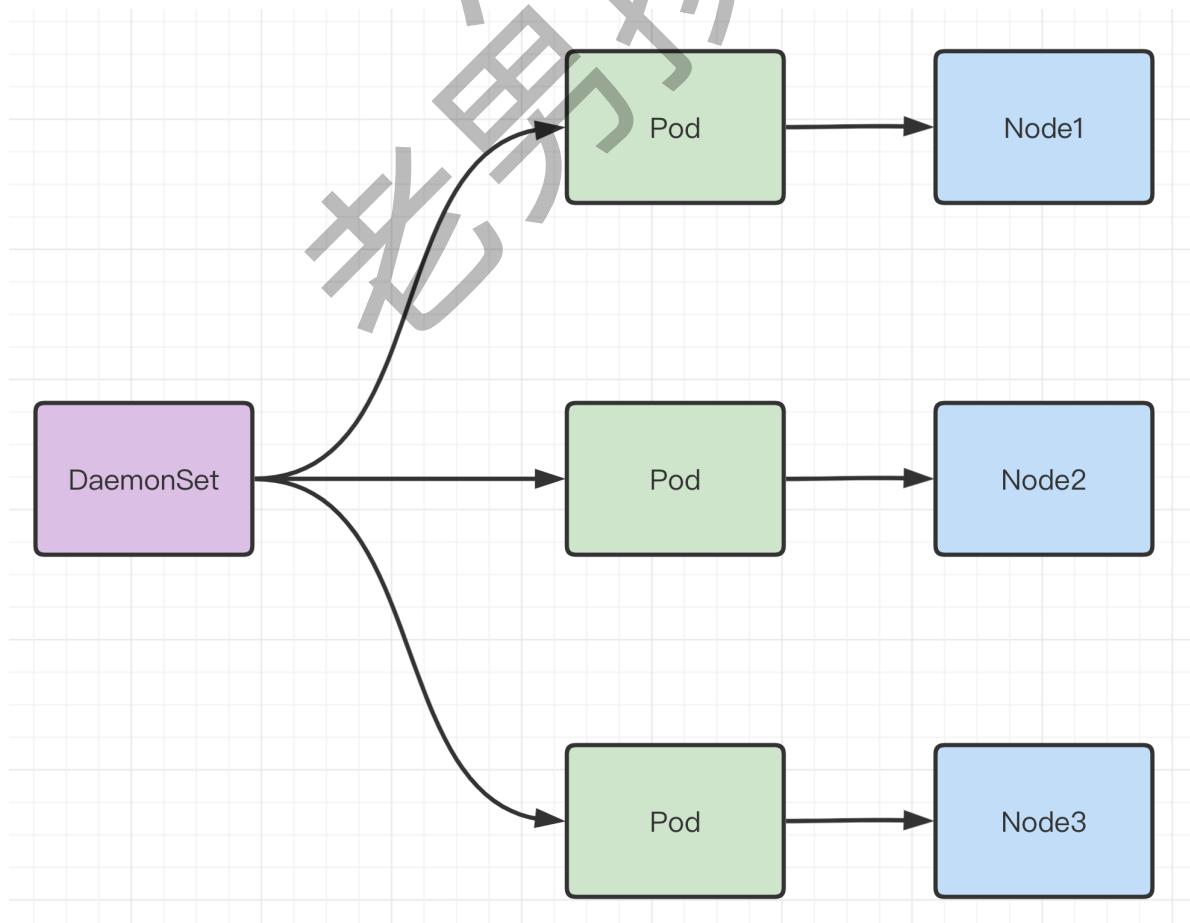
```
1 | kubectl rollout undo deployment nginx-deployment --to-revision=1
```

5.7.扩缩容

```
1 | kubectl scale deployment nginx-deployment --replicas=5  
2 | kubectl scale deployment nginx-deployment --replicas=2
```

6.DaemonSet控制器

6.1 DaemonSet类型介绍



- 1 简单来说就是每个节点部署一个POD副本
- 2 常见的应用场景：
- 3 监控容器
- 4 日志收集容器

6.2 DaemonSet举例

```
1 apiVersion: apps/v1
2 kind: DaemonSet
3 metadata:
4   name: nginx-ds
5   labels:
6     app: nginx-ds
7 spec:
8   selector:
9     matchLabels:
10    app: nginx-ds
11   template:
12     metadata:
13       labels:
14         app: nginx-ds
15     spec:
16       containers:
17         - name: nginx-ds
18           image: nginx:1.16
19           ports:
20             - containerPort: 80
```

7.HPA

7.1 HPA介绍

官网地址:

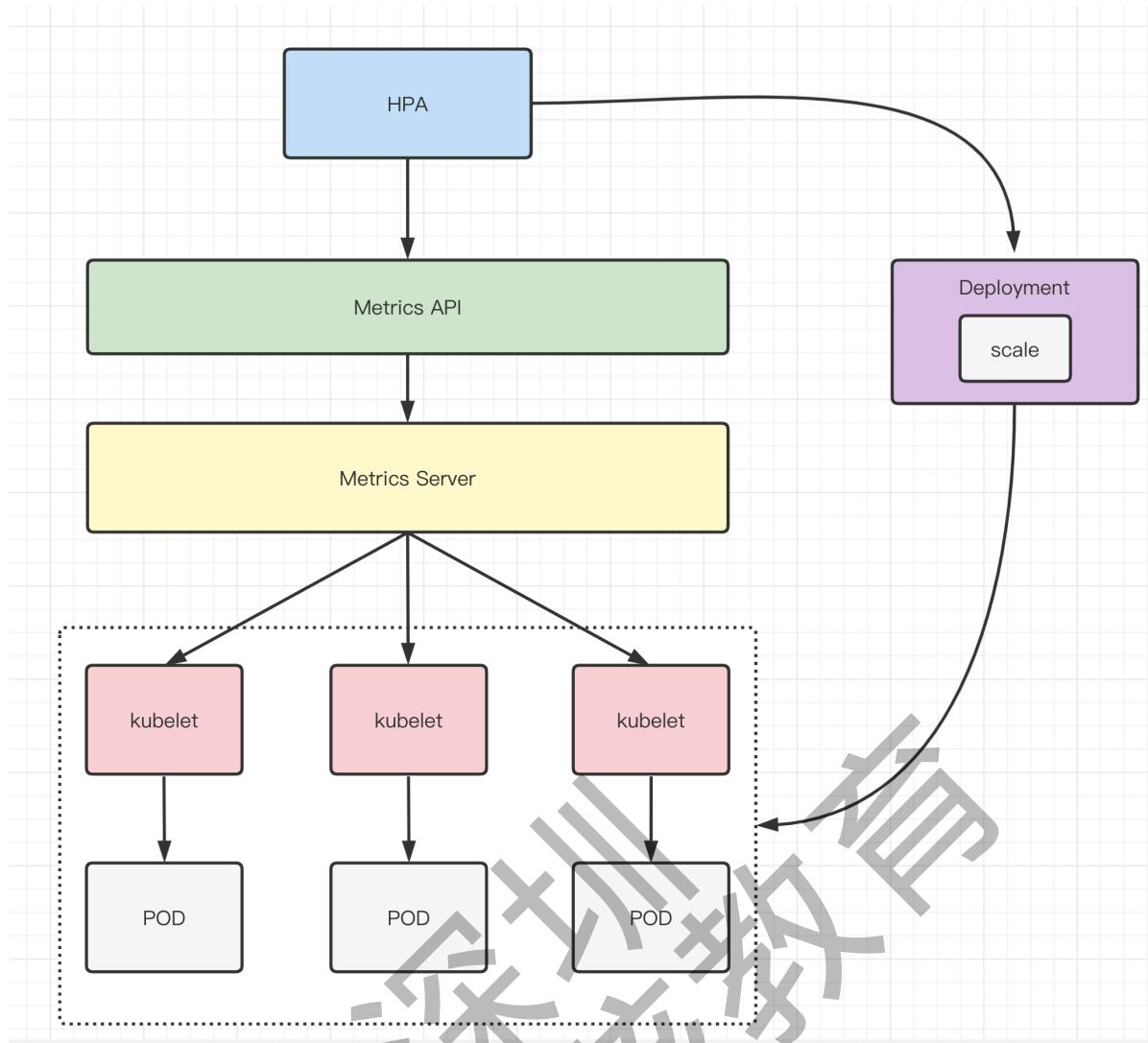
```
1 | https://kubernetes.io/zh/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/
```

HPA工作原理:

```
1 | HAP通过收集来的监控指标分析所有Pod的负载情况，并且根据我们设定好的标准来自动扩容收缩
  | ReplicationController、Deployment、ReplicaSet 或 Statefulset 中的 Pod 数量
```

7.2 Metrics Server介绍

```
1 | 在HAP早期版本使用的是一个叫Heapster组件来提供CPU和内存指标的，在后期的版本k8s转向了使用
  | Metrcis Server组件来提供Pod的CPU和内存指标，Metrcis Server通过Metrics API将数据暴露
  | 出来，然后我们就可以使用k8s的API来获取相应的数据。
```



7.3 Metrics Server安装

下载项目yaml文件

```
1 wget https://github.com/kubernetes-sigs/metrics-server/releases/download/v0.4.0/components.yaml
```

由于默认的镜像是从google云下载的，所以需要一些手段先下载下来，然后再导入本地的节点中。

修改配置文件：

```

1 spec:
2   hostNetwork: true                                #使用host网络模式
3   containers:
4     - args:
5       - --cert-dir=/tmp
6       - --secure-port=4443
7       - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
8       - --kubelet-use-node-status-port
9       - --kubelet-insecure-tls                      #跳过证书检查
10      image: metrics-server:v0.4.0                 #修改为本地的镜像
  
```

创建资源：

```
1 kubectl apply -f components.yaml
```

查看结果：

```
1 [root@master ~]# kubectl top node
2 NAME      CPU(cores)   CPU%    MEMORY(bytes)   MEMORY%
3 master     79m        7%      1794Mi        46%
4 node1      26m        2%      795Mi         20%
5 node2      23m        2%      785Mi         20%
```

7.4 生成测试镜像

创建测试首页

```
1 cat > index.php << 'EOF'
2 <?php
3     $x = 0.0001;
4     for ($i = 0; $i <= 1000000; $i++) {
5         $x += sqrt($x);
6     }
7     echo "OK!";
8 ?>
9 EOF
```

创建dockerfile

```
1 cat > dockerfile << 'EOF'
2 FROM php:5-apache
3 ADD index.php /var/www/html/index.php
4 RUN chmod a+r index.php
5 EOF
```

生成镜像

```
1 docker build -t php:v1 .
```

将镜像导出发送到其他节点：

```
1 docker save php:v1 > php.tar
2 scp php.tar 10.0.0.12:/opt/
3 scp php.tar 10.0.0.13:/opt/
```

导入镜像：

```
1 docker load < /opt/php.tar
```

7.5 创建Deployment资源

```
1 cat >php-dp.yaml << 'EOF'
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5     name: php-apache
6 spec:
7     replicas: 1
```

```
8 selector:
9   matchLabels:
10    run: php-apache
11 template:
12   metadata:
13    labels:
14     run: php-apache
15 spec:
16   containers:
17   - image: php:v1
18     imagePullPolicy: IfNotPresent
19     name: php-apache
20     ports:
21     - containerPort: 80
22       protocol: TCP
23     resources:
24       requests:
25         cpu: 200m
26 EOF
```

7.6 创建HPA资源

```
1 cat > php-hpa.yaml <<EOF
2 apiVersion: autoscaling/v1
3 kind: HorizontalPodAutoscaler
4 metadata:
5   name: php-apache
6   namespace: default
7 spec:
8   maxReplicas: 10
9   minReplicas: 1
10  scaleTargetRef:
11    apiVersion: apps/v1
12    kind: Deployment
13    name: php-apache
14    targetCPUUtilizationPercentage: 50
15 EOF
```

7.7 查看HPA扩所容情况

```
1 | kubectl get hpa -w
2 | kubectl get pod -w
```

7.8 压测

```
1 | while true; do wget -q -O- http://10.2.1.18; done
```

7.9 简单创建命令

创建dp

```
1 | kubectl run php-apache --image=php:v1 --requests(cpu=200m) --expose --port=80
```

创建hpa

```
1 | kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10
```

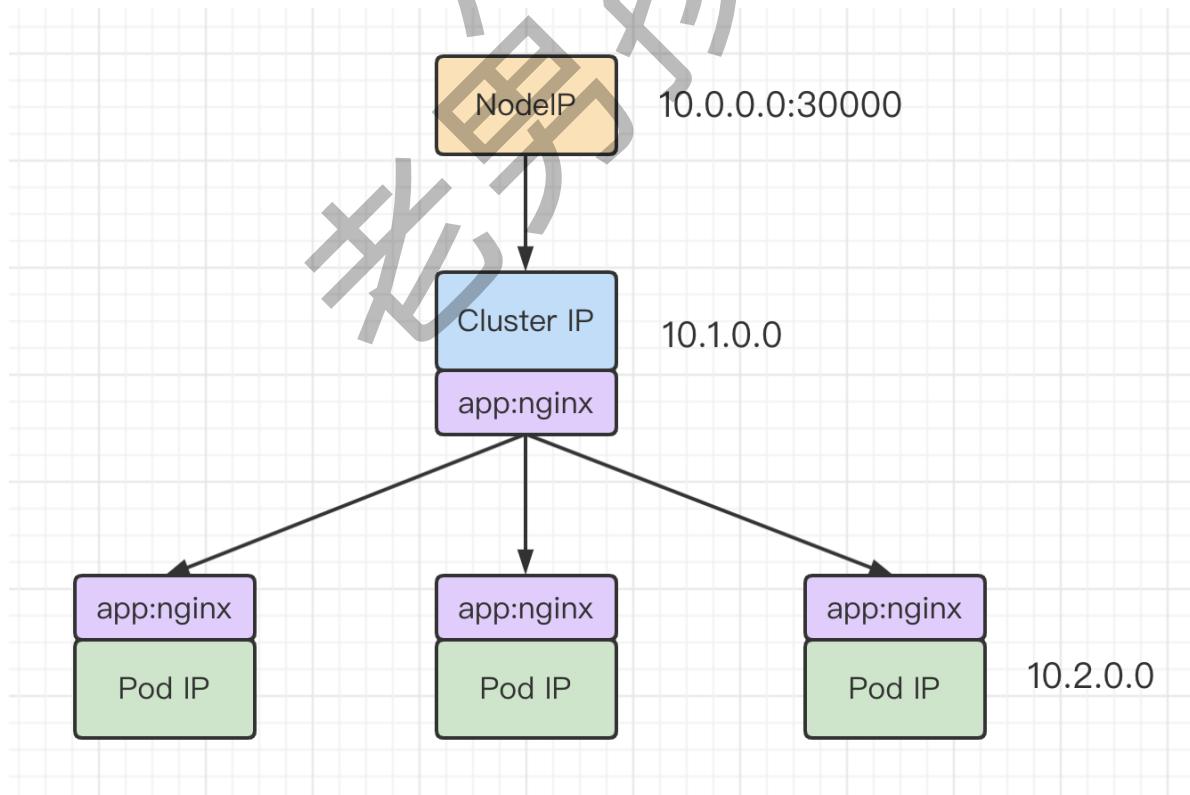
第11章 Service服务

1.Service服务介绍

- 1 通过前面的实验我们已经掌握了使用Pod控制器来管理Pod，我们也会发现，Pod的生命周期非常短暂，每次镜像升级都会销毁以及创建，而我们知道每个Pod都拥有自己的IP地址，并且随着Pod删除和创建，这个IP是会变化的。
- 2 当我们的Pod数量非常多的时候前端的入口服务该怎么知道后面都有哪些Pod呢？
- 3 为了解决这个问题k8s提供了一个对象Service和三种IP，创建的Service资源通过标签可以动态的知道后端的Pod的IP地址，在PodIP之上设计一个固定的IP，也就是ClusterIP，然后使用NodePort来对外暴露端口提供访问。
- 4 接下来我们先认识一下K8s里的三种IP及其作用。

2.k8s的三种IP

- 1 NodeIP
- 2 节点对外提供访问的IP
- 3
- 4 ClusterIP
- 5 用来动态发现和负载均衡POD的IP
- 6
- 7 PodIP
- 8 提供POD使用的IP



3.ClusterIP资源配置

资源配置清单：

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: nginx-service
5   namespace: default
6 spec:
7   selector:
8     app: nginx
9   ports:
10    - name: http
11      port: 80
12      protocol: TCP
13      targetPort: 80
14 type: ClusterIP
```

配置解释：

```
1 ports:
2 - name: http
3   port: 80          #clusterIP的端口号
4   protocol: TCP    #协议类型
5   targetPort: 80   #POD暴露的端口
6 type: ClusterIP  #service类型
```

查看ClusterIP

```
1 | kubectl get svc
```

4. NodePort资源配置

资源配置清单：

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: nginx-service
5   namespace: default
6 spec:
7   selector:
8     app: nginx
9   ports:
10    - name: http
11      port: 8080
12      protocol: TCP
13      targetPort: 80
14      nodePort: 30000
15 type: NodePort
```

配置解释：

```
1 ports:
2   - name: http
3     port: 8080      #clusterIP的端口号
4     protocol: TCP    #协议类型
5     targetPort: 80   #POD暴露的端口
6     nodePort: 30000 #NodeIP的端口号，也就是对外用户访问的端口号
7     type: NodePort    #service类型
```

查看创建的资源

```
1 | kubectl get svc
```

5.Service 服务发现

在k8s中，一个service对应的"后端"由Pod的IP和容器端口号组成，即一个完整的"IP:Port"访问地址，这在k8s里叫做Endpoint。通过查看Service的详细信息可以看到后端Endpoint的列表。

```
1 [root@node1 ~]# kubectl describe svc my-nginx
2 .....
3 Endpoints:          10.2.1.24:80,10.2.1.26:80,10.2.1.27:80 + 2 more...
```

我们也可以使用DNS域名的形式访问Service，如果在同一个命名空间里甚至可以直接使用service名来访问服务。

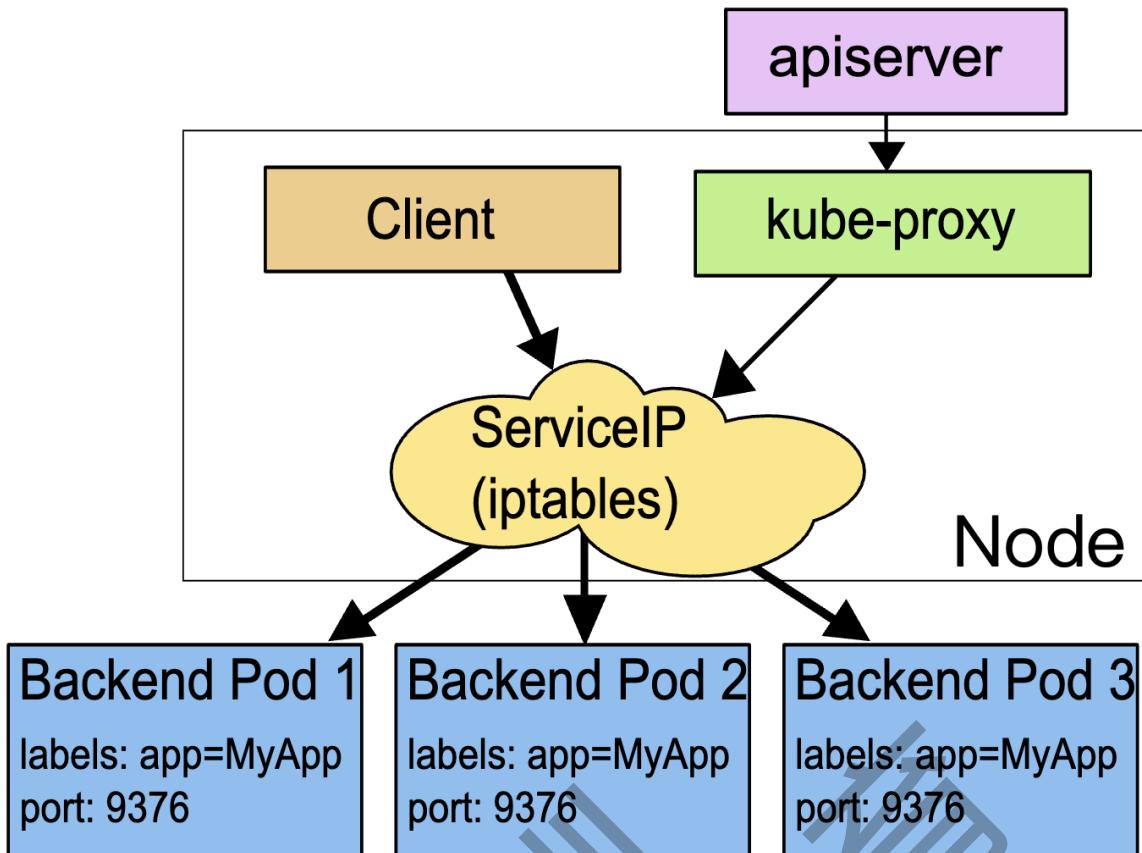
```
1 servicename.namespace.svc.cluster.local
2 servicename.namespace
3 servicename
```

6.kube-proxy负载均衡模式

通过前面的学习我们知道ClusterIP可以知道后端关联的Pod的IP，并且还可以将流量负载均衡的分摊到关联的Pod中，而这个功能就是由每个Node节点都运行的kube-proxy组件实现的。目前kube-proxy支持2种负载模式，分别为默认的iptables和性能更高的ipvs模式。

iptables模式

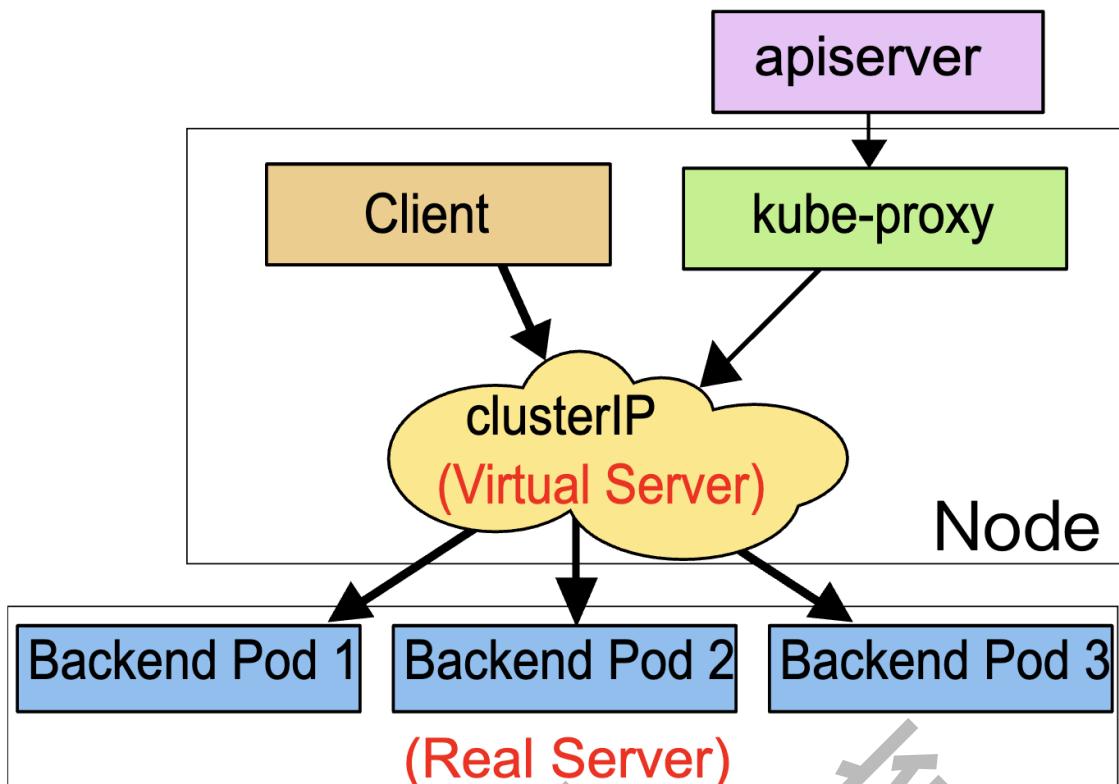
iptables模式下，kube-proxy通过部署在Linux内核的iptables规则，实现从Service到后端Endpoint列表的负载转发规则，但是如果某个后端Endpoint在转发时不可用，则请求就会失败。所以应该通过为Pod设置readinessprobe来保证只有达到ready状态的Endpoint才会被设置为Service的后端Endpoint。



ipvs模式

除了iptables模式，k8s还支持ipvs模式，ipvs在k8s 1.11版本达到Stable阶段，kube-proxy通过设置Linux内核的netlink接口设置IPVS规则，转发效率和支持的吞吐率都是最高的。ipvs模块要求Linux内核开启IPVS模块，如果操作系统未启用IPVS内核模块，kube-proxy会自动切换到iptables模式，同时，ipvs模式支持更多的负载均衡策略：

- | | |
|---|---|
| 1 | 修改默认策略需要在kube-proxy中配置 <code>-ipvs-scheduler</code> 参数来实现，但是这个参数是全局的，对所有Service类型都生效。 |
| 2 | |
| 3 | <code>rr</code> 轮询 |
| 4 | <code>lc</code> 最小连接数 |
| 5 | <code>dh</code> 目的地址哈希 |
| 6 | <code>sh</code> 源地址哈希 |
| 7 | <code>sed</code> 最短期望延时 |
| 8 | <code>nq</code> 永不排队 |



7.服务发现模式

k8s支持2种查找服务的主要模式：环境变量和DNS，前者开箱即用，而后者则需要CoreDNS组件。

环境变量模式

在一个Pod运行起来的时候，系统会自动为其容器运行环境注入所有集群中有效的Service的信息。

Service的相关信息包括服务IP，服务端口号，相关协议等。

```

1 [root@node1 ~]# kubectl exec my-nginx-7c4ff94949-6jjjq2 -- printenv|grep
2 SERVICE
3 KUBERNETES_SERVICE_PORT=443
4 KUBERNETES_SERVICE_PORT_HTTPS=443
5 MYSQL_SERVICE_PORT=3306
6 MYWEB_SERVICE_PORT=8080
7 MY_NGINX_SERVICE_PORT_HTTP=80
8 KUBERNETES_SERVICE_HOST=10.1.0.1
9 MY_NGINX_SERVICE_PORT=80
10 MYSQL_SERVICE_HOST=10.1.182.68
11 MYWEB_SERVICE_HOST=10.1.149.26

```

这样客户端就可以使用相关的环境变来访问服务了

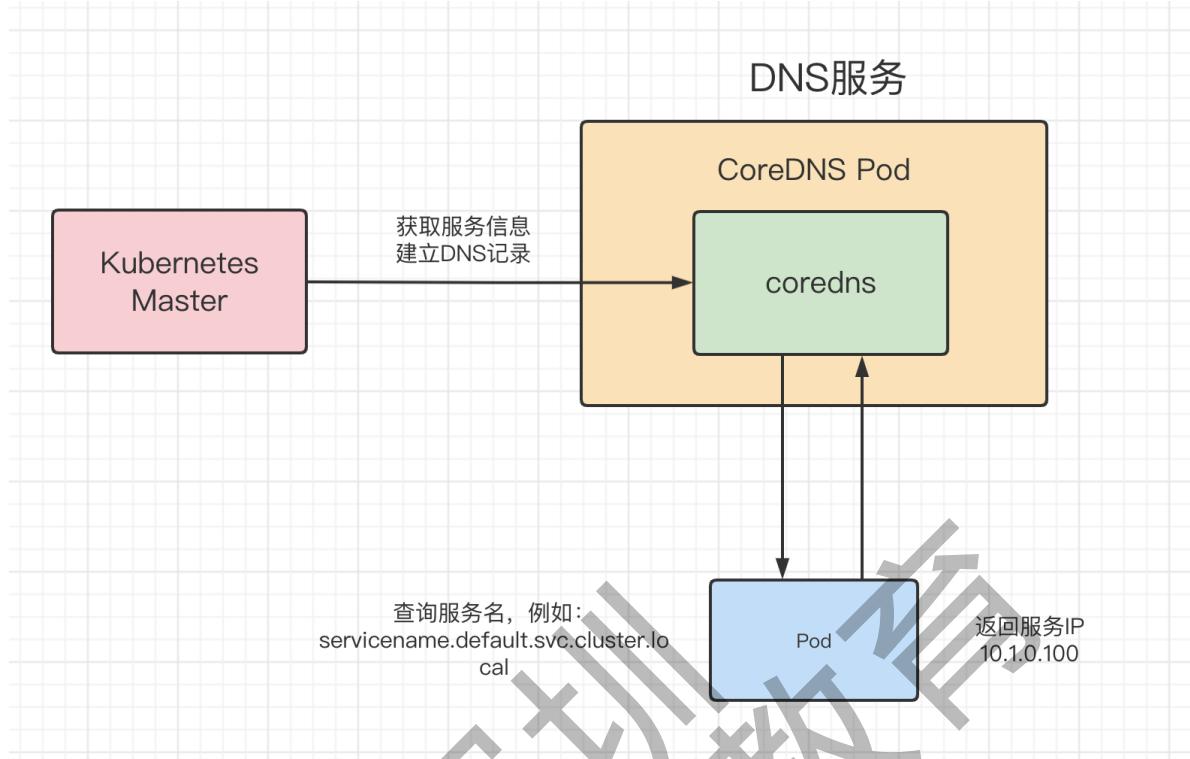
```

1 [root@node1 ~]# kubectl exec -it my-nginx-7c4ff94949-6jjjq2 -- /bin/bash
2 root@my-nginx-7c4ff94949-6jjjq2:/# curl -I ${MY_NGINX_SERVICE_HOST}
3 HTTP/1.1 200 OK

```

DNS模式

Kubernetes 提供了一个 DNS 插件 Service，当Service以DNS域名形式进行访问时，需要在k8s集群里存在一个DNS服务来完成域名到ClusterIP地址的解析工作。经过多年发展，目前由CoreDNS作为k8s集群的默认DNS服务来提供域名解析服务。



CoreDNS为其他的Pod提供Service解析服务，查看Pod的DNS解析文件可以发现nameserver是一个IP地址，这个IP地址其实是CoreDNS自己的ClusterIP。

```
1 [root@node1 ~]# kubectl -n kube-system get svc|grep kube-dns
2 kube-dns           clusterIP 10.1.0.10 <none>
3 53/UDP,53/TCP,9153/TCP
```

安装了CoreDNS的集群，当我们创建Pod后，kubelet会使用配置文件里配置的--cluster-dns=和--cluster-domain参数来传递给容器内部。kubelet配置如下；

```
1 [root@node1 ~]# cat /var/lib/kubelet/config.yaml
2 .....
3 clusterDNS:
4 - 10.1.0.10
5 clusterDomain: cluster.local
6 .....
```

查看容器的/etc/resolv.conf内容可以发现域名的相关配置

```
1 [root@node1 ~]# kubectl exec -it my-nginx-7c4ff94949-6jjq2 -- cat /etc/resolv.conf
2 nameserver 10.1.0.10
3 search default.svc.cluster.local svc.cluster.local cluster.local
4 options ndots:5
```

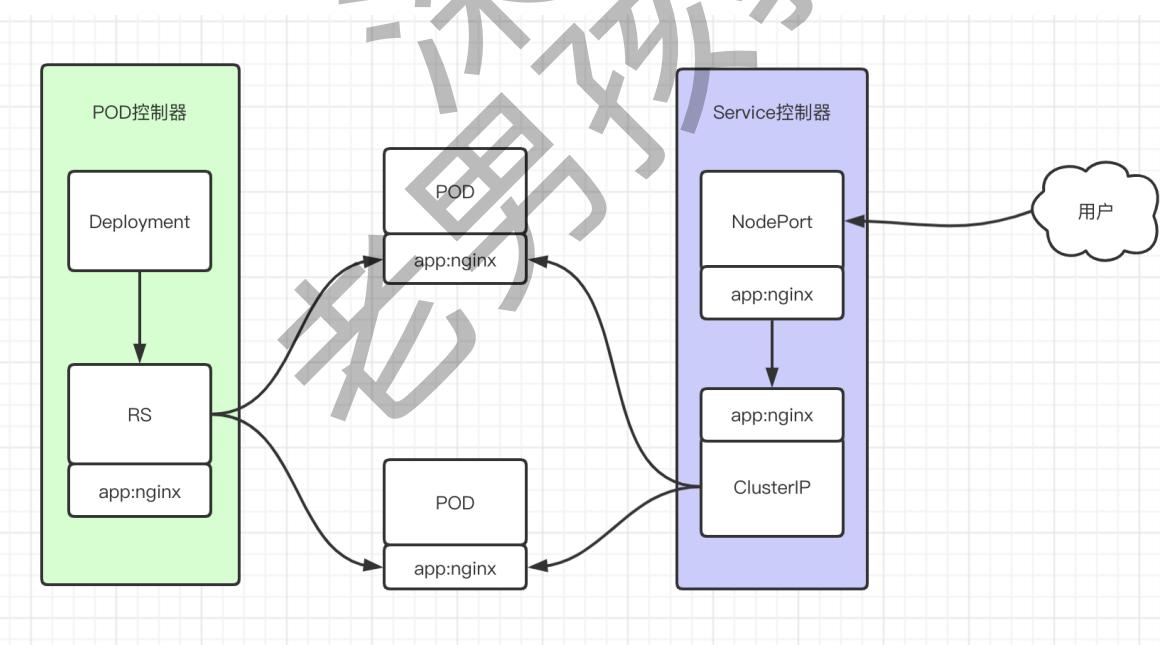
那么我们在容器里该怎么去访问Service服务呢？一共有以下几种形式。

```
1 | servicename.default.svc.cluster.local  
2 | servicename.namespace  
3 | servicename
```

实验一下：

```
1 [root@node1 ~]# kubectl exec -it my-nginx-7c4ff94949-6jjq2 -- curl -sI my-  
2 nginx.default.svc.cluster.local|head -1  
3 HTTP/1.1 200 OK  
4  
4 [root@node1 ~]# kubectl exec -it my-nginx-7c4ff94949-6jjq2 -- curl -sI my-  
5 nginx.default|head -1  
5 HTTP/1.1 200 OK  
6  
7 [root@node1 ~]# kubectl exec -it my-nginx-7c4ff94949-6jjq2 -- curl -sI my-  
8 nginx|head -1  
8 HTTP/1.1 200 OK  
9  
10 [root@node1 ~]# kubectl run -it --image busybox:1.28.3 test-dns --  
11 restart=Never -- nslookup my-nginx  
11 Server: 10.1.0.10  
12 Address 1: 10.1.0.10 kube-dns.kube-system.svc.cluster.local  
13  
14 Name: my-nginx  
15 Address 1: 10.1.182.68 my-nginx.default.svc.cluster.local
```

8.Service和Deployment关系示意图

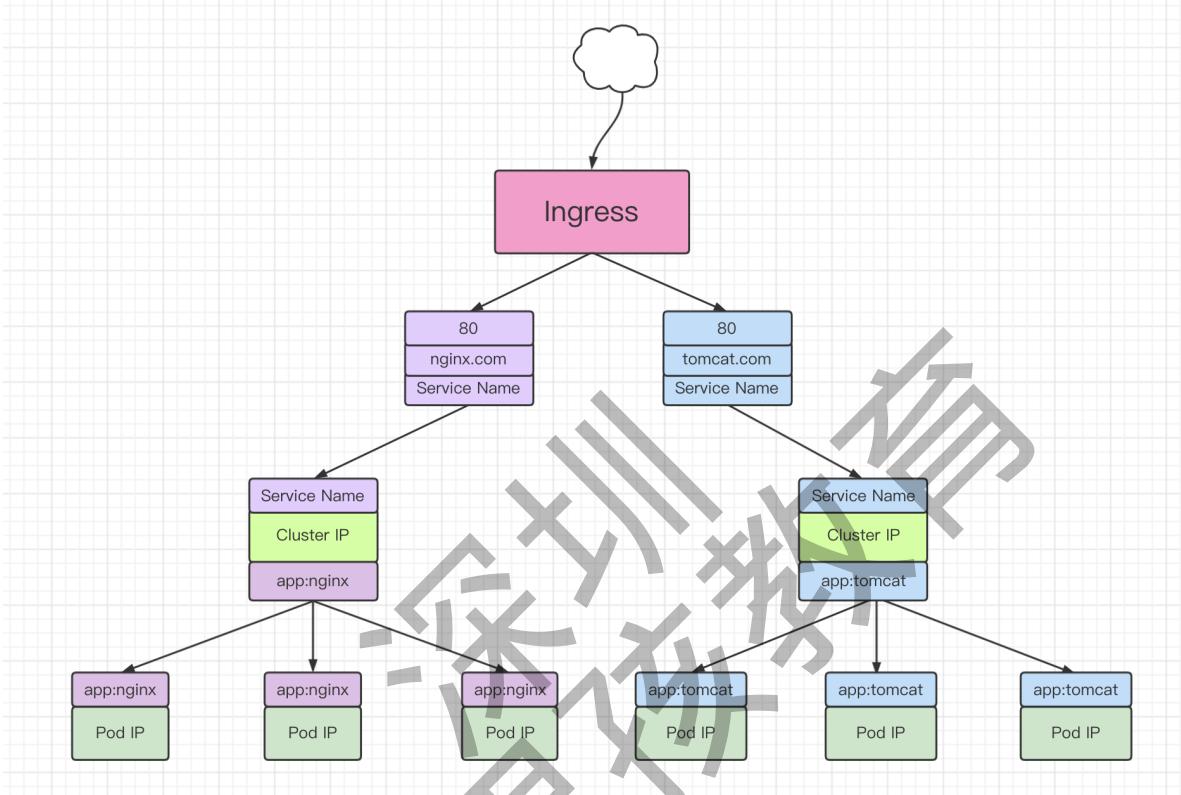


第12章 Ingress服务

1.NodePort缺点

1. 没有 ingress 之前，pod 对外提供服务只能通过 NodeIP:NodePort 的形式，但是这种形式有缺点，一个节点上的 PORT 不能重复利用。比如某个服务占用了 80，那么其他服务就不能在用这个端口了。
2. NodePort 是 4 层代理，不能解析 7 层的 http，不能通过域名区分流量
3. 为了解决这个问题，我们需要用到资源控制器叫 Ingress，作用就是提供一个统一的访问入口。工作在 7 层
4. 虽然我们可以使用 nginx/haproxy 来实现类似的效果，但是传统部署不能动态的发现我们新创建的资源，必须手动修改配置文件并重启。
5. 适用于 k8s 的 ingress 控制器主流的有 nginx-ingress 和 traefik

访问流程图：



2. 安装部署 nginx-ingress

我们可以直接使用 kubernetes 官方自带的 nginx-ingress 控制清单来部署

- 1 wget <https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/static/provider/baremetal/deploy.yaml> -o nginx-ingress.yaml

3. 修改资源配置文件

首先看下我们使用 Ingress 的架构图：

这里我们主要修改三个地方

1. Deployment 类型修改为 DaemonSet 类型
2. Pod 网络修改为 HostPort
3. 镜像地址修改为 阿里云

```
2 -----
3   323           image: registry.aliyuncs.com/google_containers/nginx-ingress-
4   controller:v0.48.1
5   -----
6   377           ports:
7     378             - name: http
8       379               containerPort: 80
9       380               protocol: TCP
10      381               hostPort: 80
11      382             - name: https
12        383               containerPort: 443
13        384               protocol: TCP
14        385               hostPort: 443
15        386             - name: webhook
16          387               containerPort: 8443
17          388               protocol: TCP
18          389               hostPort: 8443
```

3.应用资源配置

```
1 | kubectl apply -f nginx-ingress.yaml
```

4.查看创建的资源

```
1 [root@node1 nginx-ingress]# kubectl -n ingress-nginx get all
2 NAME                                         READY   STATUS    RESTARTS   AGE
3 pod/ingress-nginx-admission-create-k5rfg   0/1    Completed  0          60m
4 pod/ingress-nginx-admission-patch-kk2qb   0/1    Completed  0          60m
5 pod/ingress-nginx-controller-d44bf        1/1    Running   0          114s
6 pod/ingress-nginx-controller-rkx1m        1/1    Running   0          114s
7
8 NAME                                         TYPE        CLUSTER-IP   AGE
9 EXTERNAL-IP      PORT(S)           AGE
9 service/ingress-nginx-controller           NodePort   10.1.144.123
10 <none>          80:31053/TCP,443:30450/TCP 60m
10 service/ingress-nginx-controller-admission ClusterIP  10.1.195.159
11 <none>          443/TCP           60m
12
12 NAME                                         DESIRED   CURRENT   READY   UP-TO-
13 DATE   AVAILABLE   NODE SELECTOR           AGE
13 daemonset.apps/ingress-nginx-controller   2         2         2         2
14           kubernetes.io/os=linux        114s
15
15 NAME                                         COMPLETIONS   DURATION   AGE
16 job.batch/ingress-nginx-admission-create   1/1         51s        60m
17 job.batch/ingress-nginx-admission-patch   1/1         66s        60m
```

5.创建测试的服务

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
```

```
4   name: my-nginx
5 spec:
6   selector:
7     matchLabels:
8       app: my-nginx
9   template:
10  metadata:
11    labels:
12      app: my-nginx
13 spec:
14   containers:
15     - name: my-nginx
16       image: nginx
17       ports:
18         - containerPort: 80
19 ---
20 apiVersion: v1
21 kind: Service
22 metadata:
23   name: my-nginx
24   labels:
25     app: my-nginx
26 spec:
27   ports:
28     - port: 80
29       protocol: TCP
30       name: http
31   selector:
32     app: my-nginx
```

6. 创建Ingress规则

资源配置清单：

```
1 apiVersion: networking.k8s.io/v1
2 kind: Ingress
3 metadata:
4   name: my-nginx
5 spec:
6   rules:
7     - host: nginx.k8s.com
8       http:
9         paths:
10          - path: /
11            pathType: ImplementationSpecific
12            backend:
13              service:
14                name: my-nginx
15                port:
16                  number: 80
```

规则解释：

```

1 spec:
2   rules: #转发规则
3     - host: nginx.k8s.com #匹配的域名
4       http: #基于http协议解析
5         paths: #基于路径进行匹配
6           - path: / #匹配/路径
7             pathType: ImplementationSpecific #路径类型
8             backend: #匹配后跳转的后端服务
9               service: #设置后端跳转到Service的配置
10              name:my-nginx #跳转到名为my-nginx的ClusterIP
11              port: #跳转到的端口
12              number: 80 #Service端口号

```

pathType路径类型支持的类型：

- 1 ImplementationSpecific 系统默认，由IngressClass控制器提供
- 2 Exact 精确匹配URL路径，区分大小写
- 3 Prefix 匹配URL路径的前缀，区分大小写

7.访问测试

- 1 在windows上配置hosts解析：
- 2 nginx.k8s.com

第13章 数据持久化

1.k8s存储介绍

容器内部的的存储在生命周期是短暂的，会随着容器环境的销毁而销毁，具有不稳定性。在k8s里将对容器应用所需的存储资源抽象为存储卷(Volume)概念来解决这些问题。

k8s目前支持的Volume类型包括k8s的内部资源对象类型，开源共享存储类型和公有云存储等。分类如下：

k8s特定的资源对象：

- 1 ConfigMap 应用配置
- 2 Secret 加密数据
- 3 ServiceAccountToken token数据

k8s本地存储类型：

- 1 EmptyDir：临时存储
- 2 HostPath：宿主机目录

持久化存储(PV)和网络共享存储：

- 1 CephFS：开源共享存储系统
- 2 GlusterFS：开源共享存储系统
- 3 NFS：开源共享存储
- 4 PersistentVolumeClaim：简称PVC，持久化存储的申请空间

2.EmptyDir类型

```
1 cat >emptyDir.yaml <<EOF
2 apiVersion: v1
3 kind: Pod
4 metadata:
5   name: busybox-empty
6 spec:
7   containers:
8     - name: busybox-pod
9       image: busybox
10      volumeMounts:
11        - mountPath: /data/busybox/
12          name: cache-volume
13        command: ["/bin/sh","-c","while true;do echo $(date) >>
14          /data/busybox/index.html;sleep 3;done"]
15      volumes:
16        - name: cache-volume
17          emptyDir: {}
18 EOF
```

3.hostPath类型

3.1 type类型说明

<https://kubernetes.io/docs/concepts/storage/volumes/#hostpath>

1	DirectoryOrCreate	目录不存在就自动创建
2	Directory	目录必须存在
3	FileOrCreate	文件不存在则创建
4	File	文件必须存在

3.2 创建hostPath类型volume资源配置清单

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: busybox-nodenode
5 spec:
6   nodeName: node2
7   volumes:
8     - name: hostpath-volume
9       hostPath:
10      path: /data/node/
11      type: DirectoryOrCreate
12   containers:
13     - name: busybox-pod
14       image: busybox
15       volumeMounts:
16         - mountPath: /data/pod/
17           name: hostpath-volume
18       command: ["/bin/sh","-c","while true;do echo $(date) >>
19           /data/pod/index.html;sleep 3;done"]
```

4.NFS类型

4.1 NFS类型说明

1 我们也可以直接使用Node节点自己本身的nfs软件将共享目录挂载到Pod里，前提是NFS服务已经安装配置好，并且Node节点上安装了NFS客户端软件。

4.2 创建NFS服务

```
1 yum install nfs-utils -y
2 cat > /etc/exports << 'EOF'
3 /data/nfs-volume/blog *(rw,sync,no_root_squash)
4 EOF
5 mkdir -p /data/nfs-volume/blog
6 systemctl restart nfs
```

4.3 创建NFS类型资源清单

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: liveness-pod
5 spec:
6   nodeName: node2
7
8   volumes:
9     - name: nfs-data
10       nfs:
11         server: 10.0.0.11
12         path: /data/nfs-volume/
13   containers:
14     - name: liveness
15       image: nginx
16       imagePullPolicy: IfNotPresent
17       ports:
18         - name: http
19           containerPort: 80
20       volumeMounts:
21         - name: nfs-data
22           mountPath: /usr/share/nginx/html/
```

5.根据Node标签选择POD创建在指定的Node上

5.1 方法1: 直接选择Node节点名称

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: busybox-nodenname
5 spec:
6   nodeName: node2
7   containers:
8     - name: busybox-pod
```

```
9   image: busybox
10  volumeMounts:
11    - mountPath: /data/pod/
12      name: hostpath-volume
13    command: ["/bin/sh","-c","while true;do echo $(date) >>
14      /data/pod/index.html;sleep 3;done"]
15    volumes:
16      - name: hostpath-volume
17        hostPath:
18          path: /data/node/
19          type: DirectoryOrCreate
```

5.2 方法2: 根据Node标签选择Node节点

节点添加标签

```
1 | kubectl label nodes node3 disktype=SSD
```

资源配置清单

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: busybox-nodename
5  spec:
6    nodeSelector:
7      disktype: SSD
8    containers:
9      - name: busybox-pod
10       image: busybox
11       volumeMounts:
12         - mountPath: /data/pod/
13           name: hostpath-volume
14         command: ["/bin/sh","-c","while true;do echo $(date) >>
15           /data/pod/index.html;sleep 3;done"]
16       volumes:
17         - name: hostpath-volume
18           hostPath:
19             path: /data/node/
20             type: DirectoryOrCreate
```

6. 编写mysql的持久化deployment

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: mysql-dp
5    namespace: default
6  spec:
7    selector:
8      matchLabels:
9        app: mysql
10     replicas: 1
11     template:
12       metadata:
```

```
13     name: mysql-pod
14     namespace: default
15     labels:
16       app: mysql
17   spec:
18     volumes:
19       - name: mysql-volume
20         hostPath:
21           path: /data/mysql
22           type: DirectoryOrCreate
23     nodeSelector:
24       disktype: SSD
25     containers:
26       - name: mysql-pod
27         image: mysql:5.7
28         ports:
29           - name: mysql-port
30             containerPort: 3306
31         env:
32           - name: MYSQL_ROOT_PASSWORD
33             value: "123456"
34         volumeMounts:
35           - mountPath: /var/lib/mysql
36             name: mysql-volume
```

第14章 WordPress综合练习

使用目前所学的知识，将wordpress运行在k8s里，并且可以通过域名进行访问。

nfs准备：

```
1 yum install nfs-utils -y
2 cat > /etc/exports << 'EOF'
3 /data/nfs-volume/blog *(rw,sync,no_root_squash)
4 EOF
5 mkdir -p /data/nfs-volume/blog
6 systemctl restart nfs
```

资源配置清单：

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: mysql
5   namespace: blog
6   labels:
7     app: mysql
8 spec:
9   selector:
10    matchLabels:
11      app: mysql
12   template:
13     metadata:
14       labels:
15         app: mysql
16   spec:
```

```
17     volumes:
18     - name: db
19       hostPath:
20         path: /data/mysql
21     nodeSelector:
22       disktype: SSD
23     containers:
24     - name: mysql
25       image: mysql:5.7
26       args:
27         - --default_authentication_plugin=mysql_native_password
28         - --character-set-server=utf8mb4
29         - --collation-server=utf8mb4_unicode_ci
30     ports:
31     - name: dbport
32       containerPort: 3306
33   env:
34     - name: MYSQL_ROOT_PASSWORD
35       value: 123456
36     - name: MYSQL_DATABASE
37       value: wordpress
38     - name: MYSQL_USER
39       value: wordpress
40     - name: MYSQL_PASSWORD
41       value: wordpress
42   volumeMounts:
43     - name: db
44       mountPath: /var/lib/mysql
45 ---
46 apiVersion: v1
47 kind: Service
48 metadata:
49   name: mysql
50   namespace: blog
51 spec:
52   selector:
53     app: mysql
54   ports:
55     - name: mysqlport
56       protocol: TCP
57       port: 3306
58       targetPort: dbport
59 ---
60 apiVersion: apps/v1
61 kind: Deployment
62 metadata:
63   name: wordpress
64   namespace: blog
65   labels:
66     app: wordpress
67 spec:
68   replicas: 2
69   selector:
70     matchLabels:
71       app: wordpress
72   minReadySeconds: 5
73   strategy:
74     type: RollingUpdate
```

```
75    rollingUpdate:
76        maxSurge: 1
77        maxUnavailable: 1
78    template:
79        metadata:
80            labels:
81                app: wordpress
82        spec:
83            volumes:
84                - name: data
85                    nfs:
86                        server: 10.0.0.11
87                        path: /data/nfs-volume/blog
88            initContainers:
89                - name: init-db
90                    image: busybox
91                    command: ['sh', '-c', 'until nslookup mysql; do echo waiting for
mysql service; sleep 2; done;']
92            containers:
93                - name: wordpress
94                    image: wordpress
95                    imagePullPolicy: IfNotPresent
96            ports:
97                - name: wordpressport
98                    containerPort: 80
99            env:
100                - name: WORDPRESS_DB_HOST
101                    value: mysql:3306
102                - name: WORDPRESS_DB_USER
103                    value: wordpress
104                - name: WORDPRESS_DB_PASSWORD
105                    value: wordpress
106            readinessProbe:
107                tcpSocket:
108                    port: 80
109                    initialDelaySeconds: 5
110                    periodSeconds: 10
111            resources:
112                limits:
113                    cpu: 200m
114                    memory: 256Mi
115                requests:
116                    cpu: 100m
117                    memory: 100Mi
118            volumeMounts:
119                - name: data
120                    mountPath: /var/www/html/wp-content
121    ---
122    apiVersion: v1
123    kind: Service
124    metadata:
125        name: wordpress
126        namespace: blog
127    spec:
128        selector:
129            app: wordpress
130        ports:
131            - port: 80
```

```

132     name: wordpressport
133     protocol: TCP
134     targetPort: wdport
135   ---
136   apiVersion: networking.k8s.io/v1
137   kind: Ingress
138   metadata:
139     name: wordpress
140     namespace: blog
141   spec:
142     rules:
143       - host: blog.k8s.com
144         http:
145           paths:
146             - path: /
147               pathType: ImplementationSpecific
148             backend:
149               service:
150                 name: wordpress
151                 port:
152                   number: 80

```

第15章 持久化存储PV和PVC

1.PV和PVC介绍

PV是对底层网络共享存储的抽象，将共享存储定义为一种“资源”。

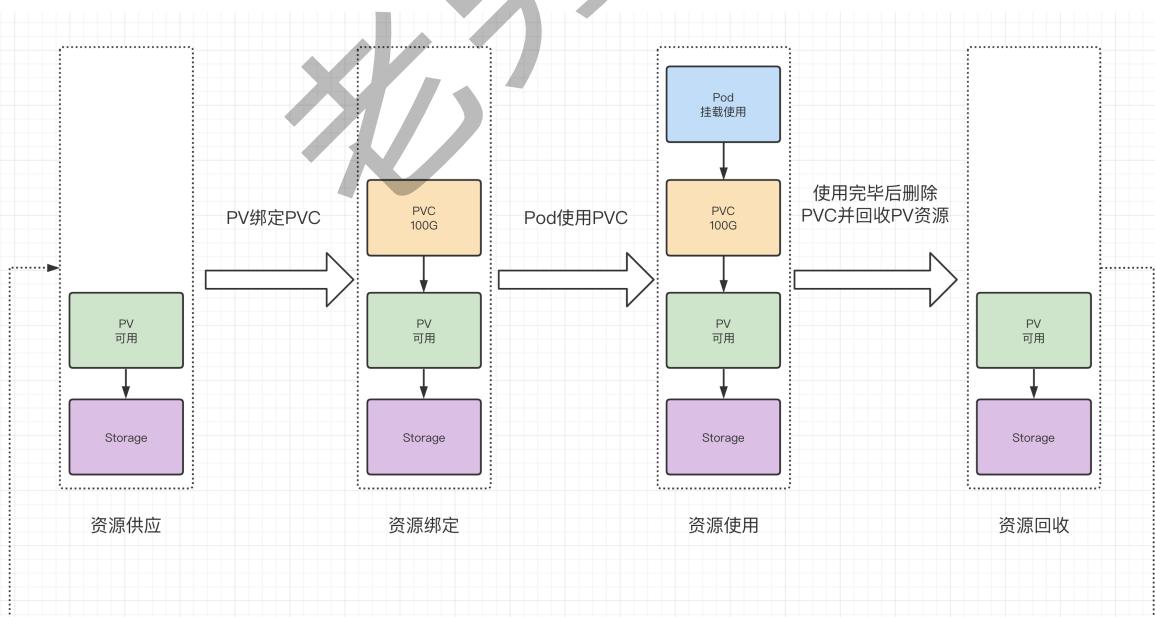
PV由管理员创建和配置

PVC则是用户对存储资源的一个“申请”。

就像Pod消费Node的资源一样，PVC能够“消费”PV资源

PVC可以申请特定的存储空间和访问模式

2.PV和PVC生命周期



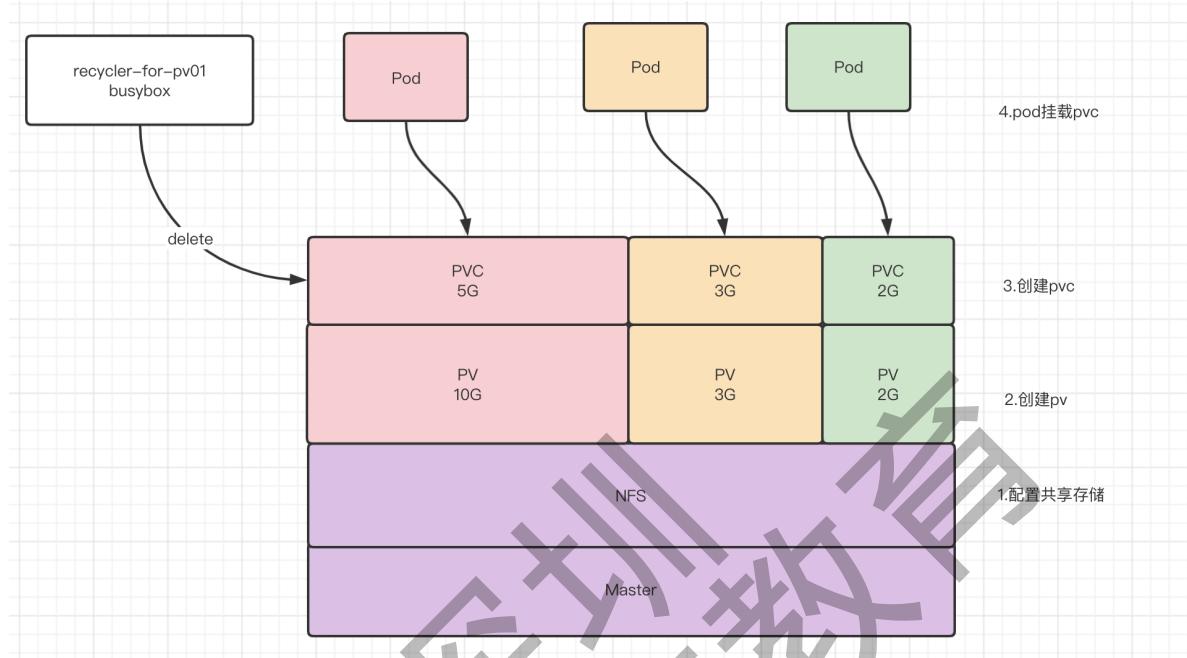
3.PV和PVC需要注意的地方

在PV的整个生命周期中，可能会处于4种不同的阶段：

- 1 Available (可用) : 表示可用状态, 还未被任何PVC绑定
- 2 Bound (可绑定) : 表示PV已经被PVC绑定
- 3 Released (已释放) : PVC被删除, 但是资源还未被集群重新声明
- 4 Failed (失败) : 表示该PV的自动回收失败

创建PVC之后, k8s就会去查找满足我们声明要求的PV, 比如storageClassName, accessModes以及容量这些是否满足要求, 如果满足要求就将PV和PVC绑定在一起。

需要注意的是目前PV和PVC之间是一对一绑定的关系, 也就是说一个PV只能被一个PVC绑定。



4. 实验-创建nfs和mysql的pv及pvc

4.1 master节点安装nfs

```

1 yum install nfs-utils -y
2 mkdir /data/nfs-volume/mysql -p
3 vim /etc/exports
4 /data/nfs-volume 10.0.0.0/24(rw,async,no_root_squash,no_all_squash)
5 systemctl restart rpcbind nfs
6 showmount -e 127.0.0.1

```

4.2 所有node节点安装nfs

```

1 yum install nfs-utils.x86_64 -y
2 showmount -e 10.0.0.11

```

4.3 编写并创建nfs-pv资源

```

1 cat >nfs-pv.yaml <<EOF
2 apiVersion: v1
3 kind: PersistentVolume
4 metadata:
5   name: pv01
6 spec:
7   capacity:

```

```

8   storage: 5Gi
9   accessModes:
10  - ReadWriteOnce
11  persistentVolumeReclaimPolicy: Recycle
12  storageClassName: nfs
13  nfs:
14    path: /data/nfs-volume/mysql
15    server: 10.0.0.10
16 EOF
17 kubectl create -f nfs-pv.yaml
18 kubectl get persistentvolume

```

PV资源关键参数:

```

1 capacity: PV存储的容量
2
3 accessModes: 访问模式,k8s支持的访问模式如下
4 -----
5 ReadWriteOnce(RWO): 读写权限, 并且只能被单个Node挂载
6 ReadOnlyMany(ROX): 只读权限, 允许被多个Node挂载
7 ReadWriteMany(RWX): 读写权限, 允许被多个Node挂载
8 -----
9
10 persistentVolumeReclaimPolicy: 回收策略
11 -----
12 Retain: 保留数据, 需要手工处理
13 Recycle: 简单清除文件的操作(例如运行rm -rf /dada/* 命令)
14 Delete: 与PV相连的后端存储完成volume的删除操作
15 目前只有NFS和HostPath两种类型的PV支持Recycle策略。
16 -----
17
18 storageClassName: 存储类别
19 具有特定类别的PV只能与请求了该类别的PVC绑定。未指定类型的PV则只能对与不请求任何类别的PVC
绑定。

```

4.4 创建mysql-pvc资源

```

1 cat >mysql-pvc.yaml <<EOF
2 apiVersion: v1
3 kind: PersistentVolumeClaim
4 metadata:
5   name: mysql-pvc
6 spec:
7   accessModes:
8     - ReadWriteOnce
9   resources:
10    requests:
11      storage: 1Gi
12    storageClassName: nfs
13 EOF
14
15 kubectl create -f mysql-pvc.yaml
16 kubectl get pvc

```

PVC资源关键参数:

```
1 accessModes: 访问模式, 与PV定义的一样  
2  
3 resources: 描述对存储资源的请求, 设置需要的存储空间大小  
4  
5 storageClassName: 存储类别, 与PV的存储类型相匹配
```

4.5 创建mysql-deployment资源

```
1 cat >mysql-dp.yaml <<EOF  
2 apiVersion: apps/v1  
3 kind: Deployment  
4 metadata:  
5   name: mysql  
6 spec:  
7   replicas: 1  
8   selector:  
9     matchLabels:  
10       app: mysql  
11   template:  
12     metadata:  
13       labels:  
14         app: mysql  
15     spec:  
16       containers:  
17         - name: mysql  
18           image: mysql:5.7  
19           ports:  
20             - containerPort: 3306  
21           env:  
22             - name: MYSQL_ROOT_PASSWORD  
23               value: "123456"  
24           volumeMounts:  
25             - name: mysql-pvc  
26               mountPath: /var/lib/mysql  
27             - name: mysql-log  
28               mountPath: /var/log/mysql  
29     volumes:  
30       - name: mysql-pvc  
31         persistentVolumeClaim:  
32           claimName: mysql-pvc  
33       - name: mysql-log  
34         hostPath:  
35           path: /var/log/mysql  
36     nodeSelector:  
37       disktype: SSD  
38 EOF  
39  
40 kubectl create -f mysql-dp.yaml  
41 kubectl get pod -o wide
```

关键参数解释：

```
1 persistentVolumeClaim: 引用PVC  
2   claimName: mysql-pvc 具体PVC的名称
```

4.6 测试方法

- 1 1. 创建nfs-pv
- 2 2. 创建mysql-pvc
- 3 3. 创建mysql-deployment并挂载mysql-pvc
- 4 4. 登陆到mysql的pod里创建一个数据库
- 5 5. 将这个pod删掉，因为deployment设置了副本数，所以会自动再创建一个新的pod
- 6 6. 登录这个新的pod，查看刚才创建的数据库是否依然能看到
- 7 7. 如果仍然能看到，则说明数据是持久化保存的

第16章 configMap资源

1.ConfigMap介绍

为什么要用configMap?

- 1 | 将配置文件和POD解耦

configMap里的配置文件是如何存储的?

- 1 | 键值对
- 2 | key:value
- 3 |
- 4 | 文件名:配置文件的内容

configMap支持的配置类型

- 1 | 直接定义的键值对
- 2 | 基于文件创建的键值对

configMap创建方式

- 1 | 命令行
- 2 | 资源配置清单

configMap的配置文件如何传递到POD里

- 1 | 变量传递
- 2 | 数据卷挂载

使用configMap的限制条件

- 1 | 1. ConfigMap必须在Pod之前创建，Pod才能引用他
- 2 | 2. ConfigMap受限于命名空间限制，只有处于同一个命名空间中的Pod才可以被引用

2.命令行创建configMap

2.1 创建命令

```
1 kubectl create configmap --help  
2  
3 kubectl create configmap nginx-config --from-literal=nginx_port=80 --from-  
literal=server_name=nginx.cookzhang.com  
4  
5 kubectl get cm  
6 kubectl describe cm nginx-config
```

关键参数：

```
1 kubectl create configmap nginx-config \创建名为nginx-  
config的configmap资源类型  
2 --from-literal=nginx_port=80 \创建变量，  
key为nginx_port, value值为80  
3 --from-literal=server_name=nginx.cookzhang.com 创建变量, key为server_name,  
value值为nginx.cookzhang.com
```

2.2 POD环境变量形式引用configMap

```
1 kubectl explain pod.spec.containers.env.valueFrom.configMapKeyRef  
2  
3 cat >nginx-cm.yaml <<EOF  
4 apiVersion: v1  
5 kind: Pod  
6 metadata:  
7   name: nginx-cm  
8 spec:  
9   containers:  
10  
11     - name: nginx-pod  
12       image: nginx:1.14.0  
13       ports:  
14         - name: http  
15           containerPort: 80  
16           env:  
17             - name: NGINX_PORT  
18               valueFrom:  
19                 configMapKeyRef:  
20                   name: nginx-config  
21                   key: nginx_port  
22             - name: SERVER_NAME  
23               valueFrom:  
24                 configMapKeyRef:  
25                   name: nginx-config  
26                   key: server_name  
27 EOF  
28 kubectl create -f nginx-cm.yaml
```

关键参数：

```
1 valueFrom: 引用变量
2 configMapKeyRef:
3   name: nginx-config 引用configmap
4     key: nginx_port 引用的configmap名称
                           应用configmap里的具体的key
```

2.3 查看pod是否引入了变量

```
1 [root@node1 ~]# kubectl exec -it nginx-cm /bin/bash
2 root@nginx-cm:~# echo ${NGINX_PORT}
3 80
4 root@nginx-cm:~# echo ${SERVER_NAME}
5 nginx.cookzhang.com
6 root@nginx-cm:~# printenv |egrep "NGINX_PORT|SERVER_NAME"
7 NGINX_PORT=80
8 SERVER_NAME=nginx.cookzhang.com
```

注意：

- 1 变量传递的形式，修改confMap的配置，POD内并不会生效
- 2 因为变量只有在创建POD的时候才会引用生效，POD一旦创建好，环境变量就不变了

3.文件形式创建configMap

3.1 创建配置文件：

```
1 cat >www.conf <<EOF
2 server {
3   listen      80;
4   server_name www.cookzy.com;
5   location / {
6     root   /usr/share/nginx/html/www;
7     index  index.html index.htm;
8   }
9 }
10 EOF
```

创建configMap资源：

```
1 kubectl create configmap nginx-www --from-file=www.conf=./www.conf
```

查看cm资源

```
1 kubectl get cm
2 kubectl describe cm nginx-www
```

3.2 编写pod并以存储卷挂载模式引用configMap的配置

```
1 cat >nginx-cm-volume.yaml <<EOF
2 apiVersion: v1
3 kind: Pod
4 metadata:
5   name: nginx-cm
```

```
6 spec:  
7   containers:  
8     - name: nginx-pod  
9       image: nginx:1.14.0  
10      ports:  
11        - name: http  
12          containerPort: 80  
13      volumeMounts:  
14        - name: nginx-www  
15          mountPath: /etc/nginx/conf.d/  
16    volumes:  
17      - name: nginx-www  
18        configMap:  
19          name: nginx-www  
20          items:  
21            - key: www.conf  
22              path: www.conf  
23 EOF
```

关键参数：

```
1 volumes: #定义存储卷  
2   - name: nginx-www #名称  
3     configMap: #引用configMap资源  
4       name: nginx-www #引用名为nginx-www的configMap资源  
5       items: #引用configMap里的具体的内容  
6         - key: www.conf #引用名为nginx-www的configMap里的key为www.conf的  
7           path: www.conf #使用key名称作为挂载文件名  
值
```

3.3 测试效果

1. 进到容器内查看文件

```
1 kubectl exec -it nginx-cm /bin/bash  
2 cat /etc/nginx/conf.d/www.conf
```

2. 动态修改

```
1 configMap  
2 kubectl edit cm nginx-www
```

3. 再次进入容器内观察配置会不会自动更新

```
1 cat /etc/nginx/conf.d/www.conf  
2 nginx -T
```

4. 配置文件内容直接以数据格式直接存储在configMap里

4.1 创建config配置清单：

```
1 cat >nginx-configMap.yaml <<EOF
2 apiVersion: v1
3 kind: ConfigMap
4 metadata:
5   name: nginx-config
6   namespace: default
7 data:
8   www.conf: |
9     server {
10       listen      80;
11       server_name www.cookzy.com;
12       location / {
13         root    /usr/share/nginx/html/www;
14         index  index.html index.htm;
15       }
16     }
17   blog.conf: |
18     server {
19       listen      80;
20       server_name blog.cookzy.com;
21       location / {
22         root    /usr/share/nginx/html/blog;
23         index  index.html index.htm;
24       }
25     }
26 E
```

4.2 应用并查看清单：

```
1 kubectl create -f nginx-configMap.yaml
2 kubectl get cm
3 kubectl describe cm nginx-config
```

4.3 创建POD资源清单并引用configMap

```
1 cat >nginx-cm-volume-all.yaml <<EOF
2 apiVersion: v1
3 kind: Pod
4 metadata:
5   name: nginx-cm
6 spec:
7   containers:
8
9     - name: nginx-pod
10       image: nginx:1.14.0
11       ports:
12         - name: http
13           containerPort: 80
14       volumeMounts:
15         - name: nginx-config
16           mountPath: /etc/nginx/conf.d/
17
18   volumes:
```

```
19 - name: nginx-config
20   configMap:
21     name: nginx-config
22     items:
23       - key: www.conf
24         path: www.conf
25       - key: blog.conf
26         path: blog.conf
27 EOF
```

4.4 应用并查看:

```
1 kubectl create -f nginx-cm-volume-all.yaml
2 kubectl get pod
3 kubectl describe pod nginx-cm
```

进入容器内并查看:

```
1 kubectl exec -it nginx-cm /bin/bash
2 ls /etc/nginx/conf.d/
3 cat /etc/nginx/conf.d/www.conf
```

4.5 测试动态修改configMap会不会生效

```
1 kubectl edit cm nginx-config
2
3 kubectl exec -it nginx-cm /bin/bash
4 ls /etc/nginx/conf.d/
5 cat /etc/nginx/conf.d/www.conf
```

第17章 k8s dashboard

1. 安装部署dashboard

官方项目地址

```
1 | https://github.com/kubernetes/dashboard
```

下载配置文件并应用

```
1 | wget
https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.4/aio/deploy/recommended.yaml
```

应用资源配置

```
1 | kubectl create -f recommended.yaml
```

查看创建的pod

```
1 [root@node1 k8s]# kubectl -n kubernetes-dashboard get pods
2 NAME                                     READY   STATUS    RESTARTS   AGE
3 dashboard-metrics-scraper-7b59f7d4df-62m6s   1/1     Running   0          62s
4 kubernetes-dashboard-665f4c5ff-4twq7        1/1     Running   0          62s
```

2. 创建Ingress资源并应用

创建资源配置：

```
1 cat > dashboard-ingress.yml << 'EOF'
2 apiVersion: networking.k8s.io/v1
3 kind: Ingress
4 metadata:
5   name: kubernetes-dashboard-ingress
6   namespace: kubernetes-dashboard
7   annotations:
8     kubernetes.io/ingress.class: "nginx"
9     ingress.kubernetes.io/ssl-passthrough: "true"
10    nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
11    nginx.ingress.kubernetes.io/rewrite-target: /
12 spec:
13   rules:
14     - host: "dashboard.k8s.com"
15       http:
16         paths:
17           - path: /
18             pathType: ImplementationSpecific
19             backend:
20               service:
21                 name: kubernetes-dashboard
22                 port:
23                   number: 443
24 EOF
```

应用配置：

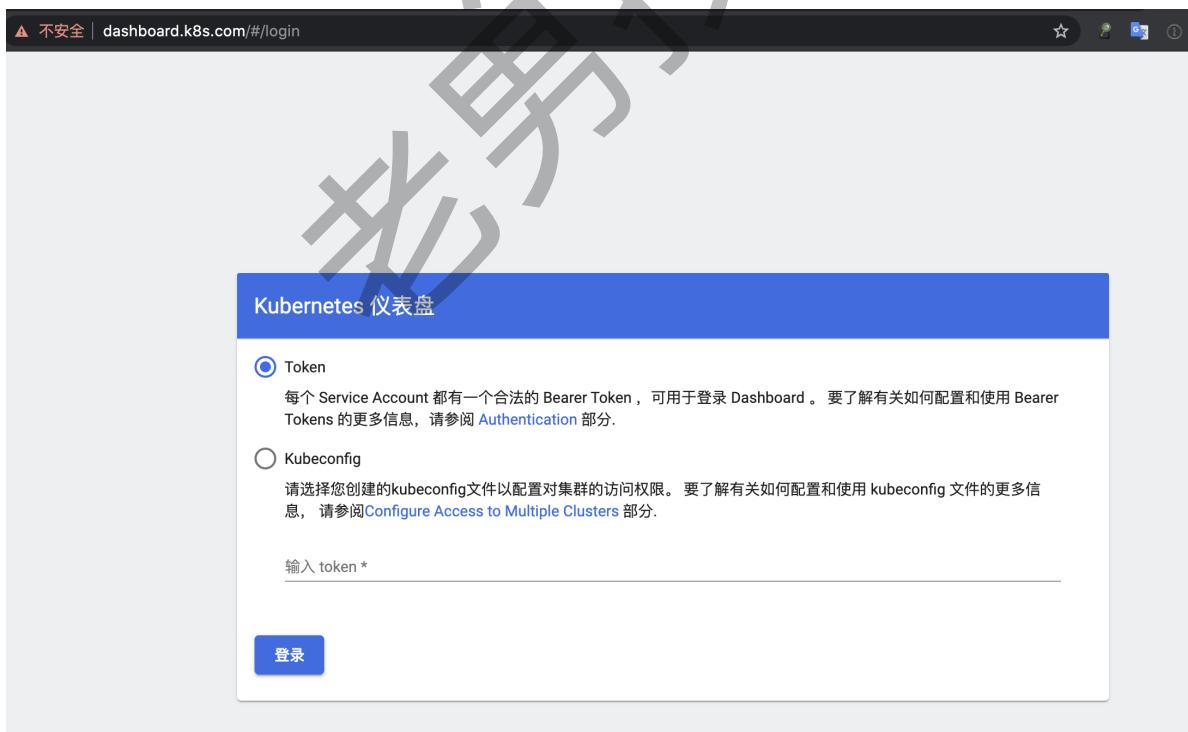
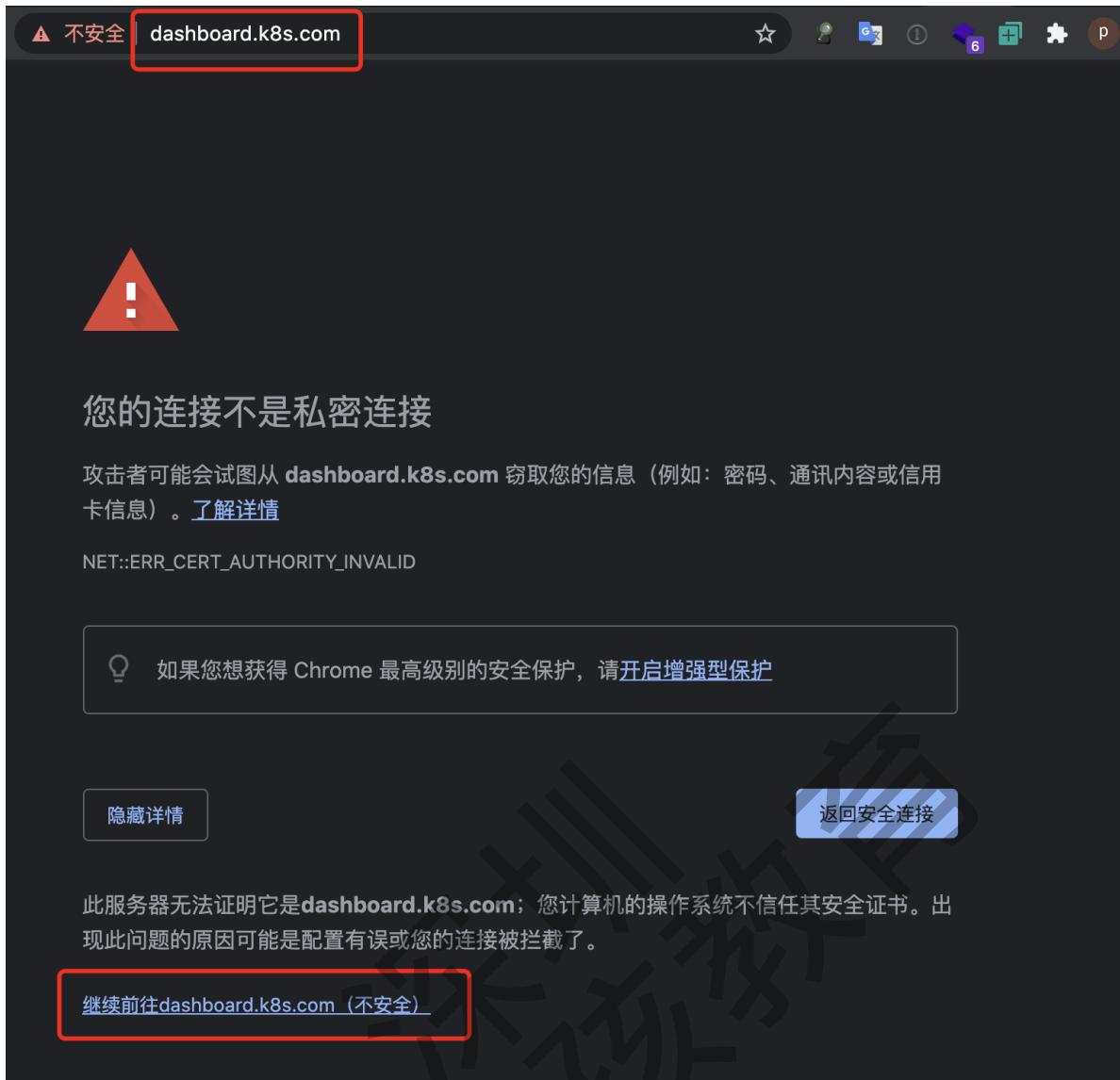
```
1 kubectl apply -f dashboard-ingress.yml
```

关键参数解释：

```
1 annotations:          #注解
2   kubernetes.io/ingress.class: "nginx"      #使用的是nginx类型的ingress
3   ingress.kubernetes.io/ssl-passthrough: "true"  #SSL透传
4   nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"  #后端使用TLS协议
5   nginx.ingress.kubernetes.io/rewrite-target: /      #URL重定向
```

3. 访问网页测试

注意要在windows上绑定hosts地址！



4. 创建管理员账户并应用

官方文档：

```
1 | https://github.com/kubernetes/dashboard/blob/master/docs/user/access-control/creating-sample-user.md
```

创建授权资源配置清单

```
1 cat > dashboard-admin.yaml<<EOF
2 apiVersion: v1
3 kind: ServiceAccount
4 metadata:
5   name: admin-user
6   namespace: kubernetes-dashboard
7 ---
8 apiVersion: rbac.authorization.k8s.io/v1
9 kind: ClusterRoleBinding
10 metadata:
11   name: admin-user
12 roleRef:
13   apiGroup: rbac.authorization.k8s.io
14   kind: ClusterRole
15   name: cluster-admin
16 subjects:
17 - kind: ServiceAccount
18   name: admin-user
19   namespace: kubernetes-dashboard
20 EOF
```

应用资源配置清单

```
1 | kubectl create -f dashboard-admin.yaml
```

5.查看资源并获取token

```
1 kubectl get pod -n kubernetes-dashboard -o wide
2 kubectl get svc -n kubernetes-dashboard
3 kubectl get secret -n kubernetes-dashboard
4 kubectl -n kubernetes-dashboard describe secret $(kubectl -n kubernetes-dashboard get secret | grep admin-user | awk '{print $1}')
```

6.浏览器访问

Kubernetes 仪表盘

Token

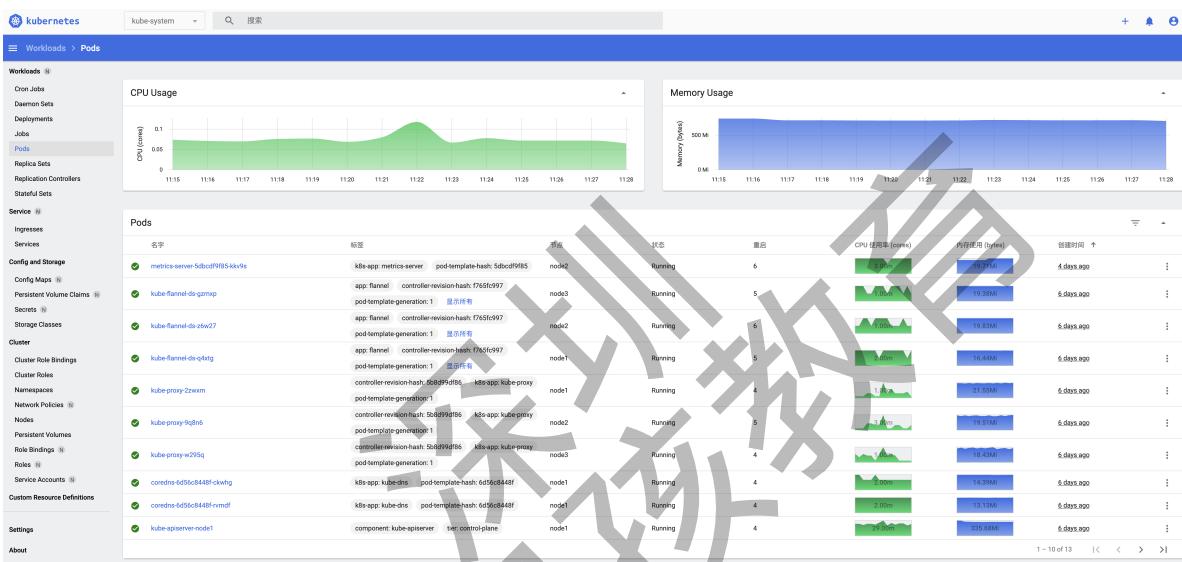
每个 Service Account 都有一个合法的 Bearer Token，可用于登录 Dashboard。要了解有关如何配置和使用 Bearer Tokens 的更多信息，请参阅 [Authentication](#) 部分。

Kubeconfig

请选择您创建的kubeconfig文件以配置对集群的访问权限。要了解有关如何配置和使用 kubeconfig 文件的更多信息，请参阅 [Configure Access to Multiple Clusters](#) 部分。

输入 token *

登录



第18章 使用harbor作为私有仓库

1.清理以前安装的Harbor

```
1 docker ps -a|grep "goharbor"|awk '{print "docker stop \"$1\""}'  
2 docker ps -a|grep "goharbor"|awk '{print "docker rm \"$1\""}'  
3 docker images|grep "goharbor"|awk '{print "docker rmi \"$1\":\"$2\""}'
```

2.安装Docker-compose

```
1 curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose  
2 chmod +x /usr/local/bin/docker-compose  
3 docker-compose --version
```

3.解压并修改harbor配置文件

```
1 | cd /opt/
2 | tar zxf harbor-offline-installer-v1.9.0-rc1.tgz
3 | cd harbor/
4 | vim harbor.yml
5 | hostname: 10.0.0.10
6 | port: 8888
7 | harbor_admin_password: 123456
8 | data_volume: /data/harbor
```

4.执行安装并访问

```
1 | ./install.sh
```

浏览器访问：

<http://10.0.0.10:8888>

5.创建一个私有仓库k8s

所有项目					
	项目名称	访问级别	角色	镜像仓库数	创建时间
<input type="checkbox"/>	k8s	私有	项目管理员	0	2021/8/7 下午10:54
1-1 共计1条记录					

6.配置docker信任仓库并重启

注意！！！三台服务器都操作!!!

```
1 | cat >/etc/docker/daemon.json<<EOF
2 | {
3 |     "registry-mirrors": ["https://ig21319y.mirror.aliyuncs.com"],
4 |     "exec-opts": ["native.cgroupdriver=systemd"],
5 |     "insecure-registries" : ["http://10.0.0.10:8888"]
6 | }
7 | EOF
8 | systemctl restart docker
```

注意！！！node1重启docker后harbor会失效，需要重启harbor

```
1 | cd /opt/harbor
2 | docker-compose stop
3 | docker-compose start
```

7.docker登陆harbor

```
1 | docker login 10.0.0.10:8888
```

8.将docker登陆凭证转化为k8s能识别的base64编码

只要一台节点操作即可

```
1 [root@node1 ~]# cat /root/.docker/config.json|base64
2 ewoJImF1dGhzIjogewoJCSIxMC4wLjAuMTE6ODg4OCI6IHsKCQkJImF1dGgioiAiWvdSdGFxNDZn
3 VE16TkRVMiIKCQl9Cg19LAoJIk0dHBIZWFkZXJzIjogewoJCSJvc2VyLUFnZW50IjogIkRvY2t1
4 ci1DbG11bnQvMTkuMDMuMTUgKGxpbnV4KSIKCX0KfQ==
```

9. 编写Secret资源配置清单

```
1 cat > harbor-secret.yaml << 'EOF'
2 apiVersion: v1
3 kind: Secret
4 metadata:
5   name: harbor-secret
6 data:
7   .dockerconfigjson:
8     ewoJImF1dGhzIjogewoJCSIxMC4wLjAuMTE6ODg4OCI6IHsKCQkJImF1dGgioiAiWvdSdGFxNDZnVE
9     16TkRVMiIKCQl9Cg19LAoJIk0dHBIZWFkZXJzIjogewoJCSJvc2VyLUFnZW50IjogIkRvY2t1ci1D
10    bG11bnQvMTkuMDMuMTUgKGxpbnV4KSIKCX0KfQ==
11 type: kubernetes.io/dockerconfigjson
12 EOF
```

10. 应用Secret资源

```
1 kubectl delete -f harbor-secret.yaml
2 kubectl create -f harbor-secret.yaml
3 kubectl get secrets
```

11. 修改镜像tag并上传到harbor

```
1 docker pull nginx:latest
2 docker tag nginx:latest 10.0.0.10:8888/k8s/nginx:latest
3 docker push 10.0.0.10:8888/k8s/nginx:latest
```

12. 修改demo资源配置清单

```
1 cat > nginx-harbor.yaml << 'EOF'
2 apiVersion: v1
3 kind: Pod
4 metadata:
5   name: demo
6   labels:
7     app: demo
8 spec:
9   imagePullSecrets:
10    - name: harbor-secret
11
12   containers:
13    - name: demo
14      image: 10.0.0.10:8888/k8s/nginx:latest
15      ports:
16        - containerPort: 80
17 EOF
```

13. 应用资源清单并查看

```
1 | kubectl apply -f nginx-harbor.yaml  
2 | kubectl get pod
```

第19章 基于角色访问控制RBAC

1. RBAC介绍

官方文档：

```
1 | https://kubernetes.io/zh/docs/reference/access-authn-authz/rbac/
```

什么是RBAC：

```
1 | RBAC 基于角色的访问控制
```

RBAC的作用是什么：

- 1 对集群中的资源和非资源权限均有完整的覆盖
- 2 RBAC的权限配置通过几个API对象即可完成，同其他API对象一样，可以用kubectl或API进行操作
- 3 可以在运行时进行调整，无需重新启动API Server

2. API 资源对象

官方文档：

```
1 | https://kubernetes.io/zh/docs/reference/using-api/#api-versioning
```

2.1 API概述

- 1 REST API 是 Kubernetes 的基本结构。
- 2 所有操作和组件之间的通信及外部用户命令都是调用 API 服务器处理的 REST API。
- 3 因此，Kubernetes 平台视一切皆为 API 对象，且它们在 API 中有相应的定义。

2.2 API版本控制

- 1 JSON 和 Protobuf 序列化模式遵循相同的模式更改原则。以下描述涵盖了这两种格式。
- 2
- 3 API 版本控制和软件版本控制是间接相关的。API 和发布版本控制提案 描述了 API 版本控制和软件版本控制间的关系。
- 4
- 5 不同的 API 版本代表着不同的稳定性和支持级别。你可以在 API 变更文档 中查看到更多的不同级别的判定标准。

版本级别说明：

- Alpha：
 - 版本名称包含 `alpha` (例如, `v1alpha1`)。
 - 软件可能会有 Bug。启用某个特性可能会暴露出 Bug。某些特性可能默认禁用。

- 对某个特性的支持可能会随时被删除，恕不另行通知。
 - API 可能在以后的软件版本中以不兼容的方式更改，恕不另行通知。
 - 由于缺陷风险增加和缺乏长期支持，建议该软件仅用于短期测试集群。
- Beta:
 - 版本名称包含 `beta`（例如，`v2beta3`）。
 - 软件被很好的测试过。启用某个特性被认为是安全的。特性默认开启。
 - 尽管一些特性会发生细节上的变化，但它们将会被长期支持。
 - 在随后的 Beta 版或稳定版中，对象的模式和（或）语义可能以不兼容的方式改变。当这种情况发生时，将提供迁移说明。模式更改可能需要删除、编辑和重建 API 对象。编辑过程可能并不简单。对于依赖此功能的应用程序，可能需要停机迁移。
 - 该版本的软件不建议生产使用。后续发布版本可能会有不兼容的变动。如果你有多个集群可以独立升级，可以放宽这一限制。

说明：请试用测试版特性时并提供反馈。特性完成 Beta 阶段测试后，就可能不会有太多的变更了。

- Stable:
 - 版本名称如 `vx`，其中 `x` 为整数。
 - 特性的稳定版本会出现在后续很多版本的发布软件中。

2.3 API 组

[API 组](#)能够简化对 Kubernetes API 的扩展。API 组信息出现在 REST 路径中，也出现在序列化对象的 `apiVersion` 字段中。

以下是 Kubernetes 中的几个组：

- 核心组的 REST 路径为 `/api/v1`。核心组并不作为 `apiVersion` 字段的一部分，例如，`apiVersion: v1`。
- 指定的组位于 REST 路径 `/apis/$GROUP_NAME/$VERSION`，并且使用 `apiVersion: $GROUP_NAME/$VERSION`（例如，`apiVersion: batch/v1`）。你可以在[Kubernetes API 参考文档](#)中查看全部的 API 组。

所有 API 组官方说明地址：

```
1 | https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.22/#-strong-api-groups-strong-
```

2.4 查看 API 组信息

我们可以使用以下 `kubectl` 命令查看 API 组的信息：

查看所有的 API 组：

```
1 | kubectl get --raw /
```

查看指定组的信息：

```
1 #默认返回的格式是没有json格式化的
2 kubectl get --raw /api/v1
3
4 #可以安装jq命令来格式化返回的json数据
5 kubectl get --raw /api/v1|jq|grep '"name"'
6
7 #查看app组信息
8 kubectl get --raw /apis/apps/v1|jq|grep '"name"'
```

3.RBAC的API资源对象

k8s里所有的资源对象都是可以被操作的，类似操作数据库一样，我们可以对资源对象执行CRUD，也就是增删改查操作。

比如以下这些资源：

```
1 Pods
2 Secrets
3 Deployment
4 Replicasets
5 Statefulsets
6 Namespaces
```

对于上面的资源可以执行以下操作：

```
1 create
2 watch
3 list
4 delete
5 get
6 edit
```

4.RBAC的基本概念

目前我们已经知道可以对资源进行操作了，但是我们还需要再了解几个概念才能在k8s创建RBAC资源。

Rule 规则

1 角色就是一组权限的集合

Role和ClusterRole 角色和集群角色

Role

1 Role定义的规则只适用单个命名空间，也就是和namespace关联的。
2 在创建Role时，必须指定该Role所属的名字空间。

ClusterRole

- 1 与之相对, `ClusterRole` 则是一个集群作用域的资源。
- 2 这两种资源的名字不同 (`Role` 和 `ClusterRole`) 是因为 Kubernetes 对象要么是名字空间作用域的, 要么是集群作用域的, 不可两者兼具。
- 3
- 4 `ClusterRole` 有若干用法。你可以用它来:
- 5 1. 定义对某个命名空间对象的访问权限, 并将在各个命名空间内完成授权;
- 6 2. 为命名空间作用域的对象设置访问权限, 并跨所有命名空间执行授权;
- 7 3. 为集群作用域的资源定义访问权限。

总结:

- 1 如果你希望在名字空间内定义角色, 应该使用 `Role`;
- 2 如果你希望定义集群范围的角色, 应该使用 `ClusterRole`。

RoleBinding和ClusterRoleBinding 角色绑定和集群角色绑定

Subject 主题

- 1 k8s里定义了3中主题资源, 分别是`user`,`group`和`Service Account`

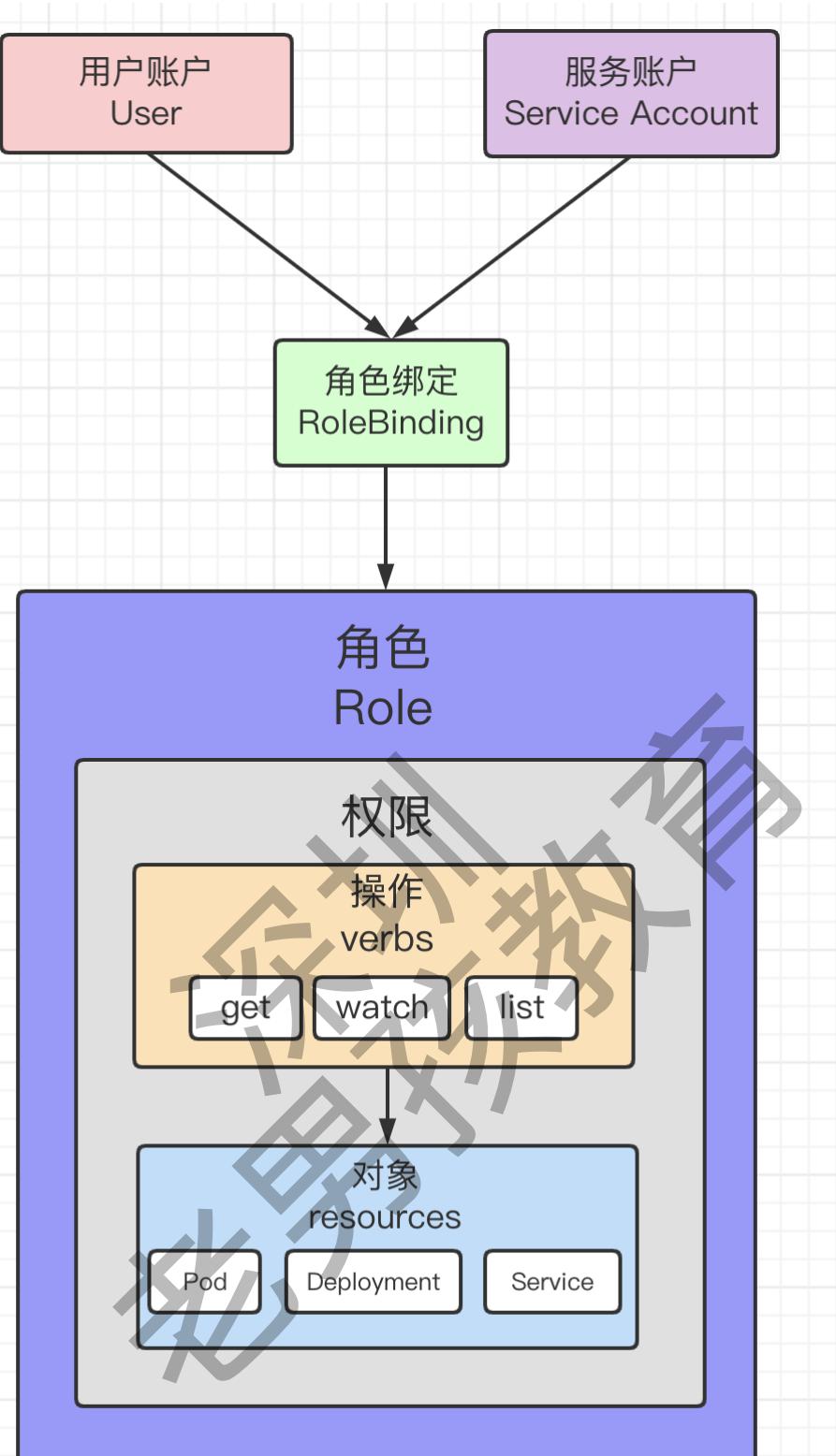
User Account和Service Account

- 1 **User Account:**
- 2 用户账号, 用户是外部独立服务管理的, 不属于k8s内部的API
- 3
- 4 **ServiceAccount:**
- 5 也是一种账号, 但他并不是给k8s集群的用户用的, 而是给运行Pod里的进程用的, 他为Pod里的进程提供了必要的身份证明。

RoleBinding和ClusterRoleBinding

- 1 角色绑定 (`Role Binding`) 是将角色中定义的权限赋予一个或者一组用户。
- 2 它包含若干 主体 (用户、组或服务账户) 的列表和对这些主体所获得的角色的引用。
- 3
- 4 `RoleBinding` 在指定的名字空间中执行授权,
- 5 `ClusterRoleBinding` 在集群范围执行授权。

RBAC关系图



5. RBAC 举例

5.1 创建 Role

创建一个位于default命名空间的Role，拥有对pods的访问权限

```
1 apiVersion: rbac.authorization.k8s.io/v1
2 kind: Role
3 metadata:
4   namespace: default
5   name: pod-reader
6 rules:
7 - apiGroups: [""]
8   resources: ["pods"]
9   verbs: ["get", "watch", "list"]
```

参数解释:

```
1 rules:
2 - apiGroups: [""]
3   resources: ["pods"]
4   verbs: ["get", "watch", "list"]
```

#""表示核心 API组
#需要操作的资源对象列表
#允许对资源对象操作的方法列表

5.2 创建RoleBinding

```
1 apiVersion: rbac.authorization.k8s.io/v1
2 kind: RoleBinding
3 metadata:
4   name: read-pods
5   namespace: default
6 subjects:
7 - kind: User
8   name: oldboy
9   apiGroup: rbac.authorization.k8s.io
10 roleRef:
11   kind: Role
12   name: pod-reader
13   apiGroup: rbac.authorization.k8s.io
```

参数解释:

```
1 subjects:
2 - kind: User
3   name: oldboy
4   apiGroup: rbac.authorization.k8s.io
5 roleRef:
6   kind: Role
7   name: pod-reader
8   apiGroup: rbac.authorization.k8s.io
```

5.3 创建 ClusterRole

```
1 apiVersion: rbac.authorization.k8s.io/v1
2 kind: ClusterRole
3 metadata:
4   # "namespace" 被忽略, 因为 clusterRoles 不受名字空间限制
5   name: secret-reader
6 rules:
7 - apiGroups: [""]
8   # 在 HTTP 层面, 用来访问 Secret 对象的资源的名称为 "secrets"
9   resources: ["secrets"]
10  verbs: ["get", "watch", "list"]
```

5.4 创建 ClusterRoleBinding

```
1 apiVersion: rbac.authorization.k8s.io/v1
2 # 此集群角色绑定允许 "manager" 组中的任何人访问任何名字空间中的 secrets
3 kind: ClusterRoleBinding
4 metadata:
5   name: read-secrets-global
6 subjects:
7 - kind: Group
8   name: manager # 'name' 是区分大小写的
9   apiGroup: rbac.authorization.k8s.io
10 roleRef:
11   kind: ClusterRole
12   name: secret-reader
13   apiGroup: rbac.authorization.k8s.io
```

5.5 创建只能访问某个namespace的普通用户

创建用户凭证

```
1 openssl genrsa -out oldboy.key 2048
2 openssl req -new -key oldboy.key -out oldboy.csr -subj "/CN=oldboy/O=oldboy"
3 openssl x509 -req -in oldboy.csr -CA /etc/kubernetes/pki/ca.crt -CAkey
/etc/kubernetes/pki/ca.key -CAcreateserial -out oldboy.crt -days 300
4 kubectl config set-credentials oldboy --client-certificate=oldboy.crt --
client-key=oldboy.key
5 kubectl config set-context oldboy-context --cluster=kubernetes --
namespace=kube-system --user=oldboy
6 kubectl get pods --context=oldboy-context
```

创建角色

```

1 cat > oldboy-role.yaml << 'EOF'
2 apiVersion: rbac.authorization.k8s.io/v1
3 kind: Role
4 metadata:
5   name: oldboy-role
6   namespace: kube-system
7 rules:
8 - apiGroups: [ "", "apps" ]
9   resources: ["deployments", "replicasets", "pods"]
10  verbs: ["*"]
11 EOF
12 kubectl apply -f oldboy-role.yaml

```

创建角色绑定

```

1 cat > oldboy-rolebinding.yml << 'EOF'
2 apiVersion: rbac.authorization.k8s.io/v1
3 kind: RoleBinding
4 metadata:
5   name: oldboy-rolebinding
6   namespace: kube-system
7 subjects:
8 - kind: User
9   name: oldboy
10  apiGroup: ""
11 roleRef:
12   kind: Role
13   name: oldboy-role
14   apiGroup: rbac.authorization.k8s.io
15 EOF
16 kubectl apply -f oldboy-rolebinding.yml

```

测试访问

	NAME					READY
	STATUS	RESTARTS	AGE			
1	[root@node1 ~]# kubectl get pods --context=oldboy-context					
2	coredns-6d56c8448f-ckwhg		1/1	Running	4	7d19h
3	coredns-6d56c8448f-rvmdf		1/1	Running	4	7d19h
4	etcd-node1		1/1	Running	4	7d19h
5	kube-apiserver-node1		1/1	Running	4	7d19h
6	kube-controller-manager-node1		1/1	Running	14	7d19h
7	kube-flannel-ds-gzmxp		1/1	Running	5	7d19h
8	kube-flannel-ds-q4xtg		1/1	Running	5	7d19h
9	kube-flannel-ds-z6w27		1/1	Running	6	7d19h
10	kube-proxy-2zwxm		1/1	Running	4	7d19h
11	kube-proxy-9q8n6		1/1	Running	5	7d19h
12	kube-proxy-w295q		1/1	Running	4	7d19h
13	kube-scheduler-node1		1/1	Running	15	7d19h
14	metrics-server-5dbcdf9f85-kkv9s		1/1	Running	6	5d11h

访问其他的命名空间

```
1 [root@node1 ~]# kubectl --context=oldboy-context get pods --namespace=default
2 Error from server (Forbidden): pods is forbidden: user "cnych" cannot list
  resource "pods" in API group "" in the namespace "default"
```

5.6 只能访问某个 namespace 的 ServiceAccount

创建ServiceAccount凭证

```
1 cat > oldboy-sa.yml << 'EOF'
2 apiVersion: v1
3 kind: ServiceAccount
4 metadata:
5   name: oldboy-sa
6   namespace: kube-system
7 EOF
8 kubectl apply -f oldboy-sa.yml
```

创建角色

```
1 cat > oldboy-sa-role.yml << 'EOF'
2 apiVersion: rbac.authorization.k8s.io/v1
3 kind: Role
4 metadata:
5   name: oldboy-sa-role
6   namespace: kube-system
7 rules:
8 - apiGroups: []
9   resources: ["pods"]
10  verbs: ["get", "watch", "list"]
11 - apiGroups: ["apps"]
12   resources: ["deployments"]
13   verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
14 EOF
15 kubectl apply -f oldboy-sa-role.yml
```

创建角色绑定

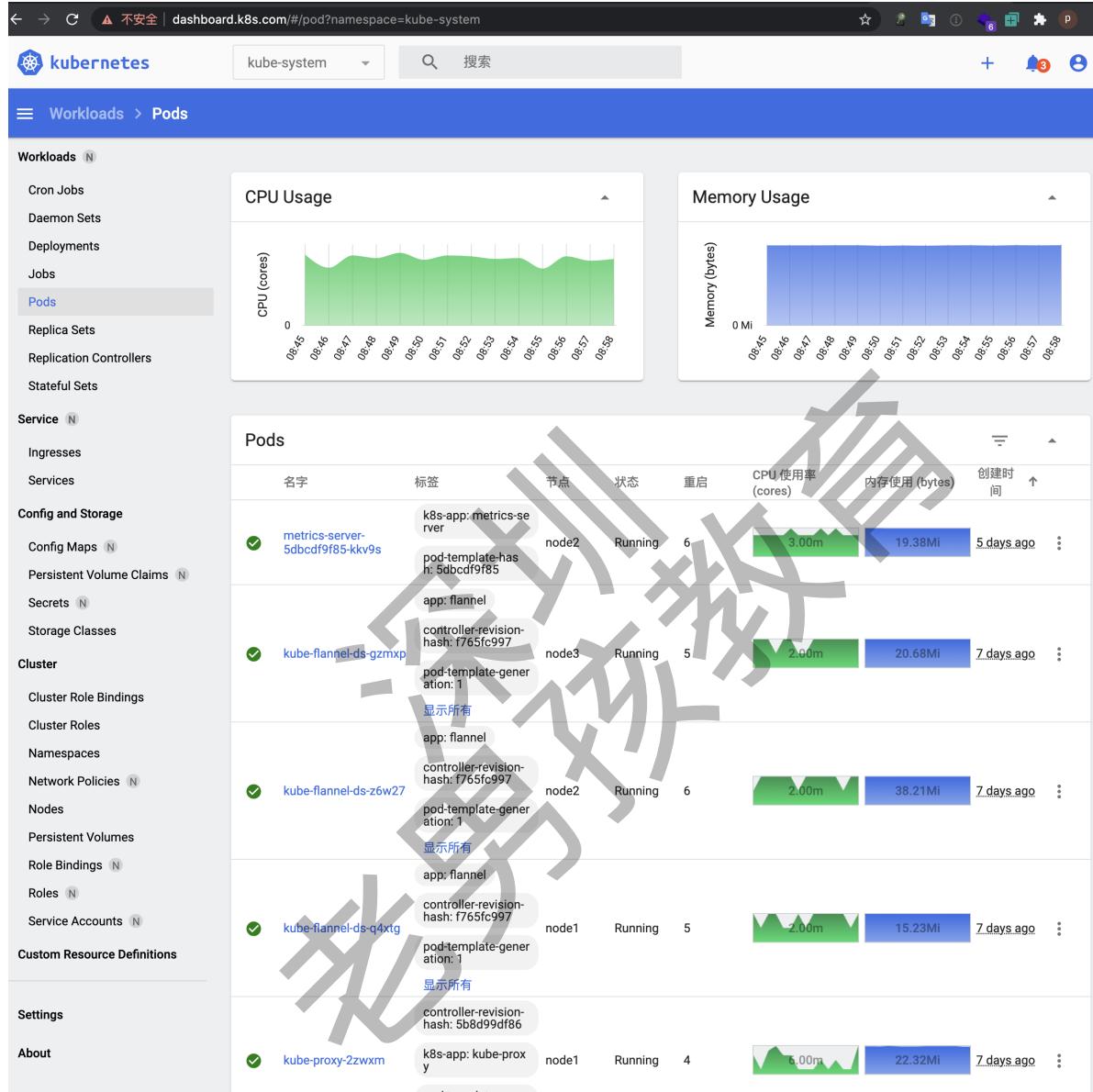
```
1 cat > oldboy-sa-rolebinding.yml << 'EOF'
2 kind: RoleBinding
3 apiVersion: rbac.authorization.k8s.io/v1
4 metadata:
5   name: oldboy-sa-rolebinding
6   namespace: kube-system
7 subjects:
8 - kind: ServiceAccount
9   name: oldboy-sa
10  namespace: kube-system
11 roleRef:
12   kind: Role
13   name: oldboy-sa-role
14   apiGroup: rbac.authorization.k8s.io
15 EOF
16 kubectl apply -f oldboy-sa-rolebinding.yml
```

获取用户的token

```
1 | kubectl -n kubernetes-dashboard describe secret $(kubectl -n kubernetes-dashboard get secret | grep oldboy-sa | awk '{print $1}')
```

dashboard测试访问

注意选择命名空间为kube-system



第20章 Pod调度

官方文档：

```
1 | https://kubernetes.io/zh/docs/concepts/scheduling-eviction/assign-pod-node/#node-affinity
```

1.nodeSelector

- 1 nodeSelector 是节点选择约束的最简单推荐形式。
- 2 nodeSelector 是 PodSpec 的一个字段。 它包含键值对的映射。
- 3 为了使 pod 可以在某个节点上运行，该节点的标签中 必须包含这里的每个键值对（它也可以具有其他标签）。
- 4 最常见的用法的是一对键值对

2.亲属性和反亲属性

节点亲属性 nodeAffinity

亲属性可以分为软需求和硬需求两种：

- 硬需求：必须满足指定的规则才可以调度Pod到Node上(功能和nodeSelector很像，但是使用的是不同的语法)，相当于硬设置。
- 软需求：强调优先满足指定规则，调度器会尝试调度Pod到Node上，但并不强求，相当于软限制。多个优先级规则还可以设置权重(weight)用来定义执行的先后顺序。

Pod亲属性 podAffinity

Pod反亲属性 podAntiAffinity

3.污点与容忍

第21章 k8s包管理Helm

第22章 k8s日志收集

第23章 k8s代码上线

第24章 prometheus

1.官网地址

- 1 <https://github.com/prometheus/prometheus>

2.监控k8s需要的组件

- 1 使用metric-server收集数据给k8s集群内使用，如kubectl, hpa, scheduler等
- 2 使用prometheus-operator部署prometheus，存储监控数据
- 3 使用kube-state-metrics收集k8s集群内资源对象数据
- 4 使用node_exporter收集集群中各节点的数据
- 5 使用prometheus收集apiserver, scheduler, controller-manager, kubelet组件数据
- 6 使用alertmanager实现监控报警
- 7 使用grafana实现数据可视化
- 8
- 9 metrics-server 主要关注的是资源度量 API 的实现，比如 CPU、文件描述符、内存、请求延时等指标。
- 10 kube-state-metrics 主要关注的是业务相关的一些元数据，比如 Deployment、Pod、副本状态等

第25章 k8s集群维护

第26章 k8s二进制安装部署

第27章 k8s在阿里云的使用

深入学习云计算