目录

CKA2-k8s 企业运维+落地实战

Docker 官网：https://docs.docker.com/

Docker 的 github 地址：https://github.com/moby/moby

Dockerhub 官网：https://registry.hub.docker.com

博客：https://www.cnblogs.com/KrillLiszt/

百度网盘链接：https://pan.baidu.com/s/15t_TSH5RRpCFXV-93JHpNw?pwd=8od3

1 kubeadm 安装 k8s 1.25 单 master 集群

1.1 初始化安装 k8s 集群的实验环境

K8S 集群角色	IP	主机名	安装的组件
控制节点	192.168.40.180	master1	apiserver 、 controller-manager 、 schedule、kubelet、etcd、kube-proxy、容器运行时、calico、keepalived、nginx
工作节点	192.168.40.181	node1	Kube-proxy、calico、coredns、容器运行时、kubelet
工作节点	192.168.40.182	node2	Kube-proxy、calico、coredns、容器运行时、kubelet

初始化操作三个节点均需要操作：

1.1.1 修改机器 ip

```
# vim /etc/sysconfig/network-scripts/ifcfg-eth0
NAME=eth0
DEVICE=eth0
ONBOOT=yes
TYPE=Ethernet
BOOTPROTO=static
IPADDR=192.168.40.180
NETMASK=255.255.255.0
GATEWAY=192.168.40.253
DEFROUTE=yes
```

1.1.2 关闭 selinux

```
# sed -i 's/SELINUX=enforcing/SELINUX=disabled/g' /etc/selinux/config
```

1.1.3 配置主机名称

```
# hostnamectl set-hostname docker && bash
```

1.1.4 关闭交换分区 swap

```
# vim /etc/fstab   //注释 swap 挂载，给 swap 这行开头加一下注释。
#/dev/mapper/centos-swap swap  swap  defaults  0  0
```

1.1.5 修改机器内核参数

```
# modprobe br_netfilter
# echo "modprobe br_netfilter" >> /etc/profile
# cat > /etc/sysctl.d/k8s.conf <<EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
```

```
net.ipv4.ip_forward = 1
EOF
# sysctl -p /etc/sysctl.d/k8s.conf
# vim /etc/rc.sysinit    //重启后模块失效，下面是开机自动加载模块的脚本，在/etc/新建 rc.sysinit 文件
#!/bin/bash
for file in /etc/sysconfig/modules/*.modules; do
[ -x $file ] && $file
done
# vim /etc/sysconfig/modules/br_netfilter.modules    //在/etc/sysconfig/modules/目录下新建文件
modprobe br_netfilter
# chmod 755 /etc/sysconfig/modules/br_netfilter.modules    //增加权限
```

1.1.6 关闭防火墙

```
# systemctl stop firewalld; systemctl disable firewalld
```

1.1.7 配置 yum 源

备份基础 repo 源

```
# mkdir /root/repo.bak
# cd /etc/yum.repos.d/
# mv * /root/repo.bak/
```

下载阿里云的 repo 源，把 CentOS-Base.repo 和 epel.repo 文件上传到 master1 主机的/etc/yum.repos.d/目录下

配置国内阿里云 docker 的 repo 源

```
# yum -y install yum-utils
# yum-config-manager --add-repo http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo
```

1.1.8 配置时间同步

```
# yum -y install ntpdate    //安装 ntpdate 命令
# ntpdate cn.pool.ntp.org    //跟网络时间做同步
# crontab -e    //把时间同步做成计划任务
* */1 * * * /usr/sbin/ntpdate cn.pool.ntp.org
# service crond restart    //重启 crond 服务
```

1.1.9 安装基础软件包

```
# yum -y install yum-utils openssh-clients device-mapper-persistent-data lvm2 wget net-tools nfs-utils lrzsz gcc gcc-c++ make cmake libxml2-devel openssl-devel curl curl-devel unzip sudo ntp libaio-devel vim ncurses-devel autoconf automake zlib-devel  python-devel epel-release openssh-server socat  ipvsadm conntrack ntpdate telnet ipvsadm
```

1.2 安装 containerd 及 docker 服务

部署服务的操作三个节点均需要操作：

1.2.1 安装 containerd

```
# yum install containerd.io-1.6.6 -y
```

1.2.2 配置 containerd 配置

```
# mkdir -p /etc/containerd
# containerd config default > /etc/containerd/config.toml    //生成 containerd 配置文件
# vim /etc/containerd/config.toml    //修改配置文件
```

把 SystemdCgroup = false 修改成 SystemdCgroup = true

把 sandbox_image = "k8s.gcr.io/pause:3.6"修改成 sandbox_image =

"registry.aliyuncs.com/google_containers/pause:3.7"

1.2.3 配置 containerd 开机启动，并启动 containerd

```
# systemctl enable containerd  --now
```

1.2.4 修改/etc/crictl.yaml 文件

```
# cat > /etc/crictl.yaml <<EOF
runtime-endpoint: unix:///run/containerd/containerd.sock
image-endpoint: unix:///run/containerd/containerd.sock
timeout: 10
debug: false
EOF
# systemctl restart containerd
```

1.2.5 配置 containerd 镜像加速器

```
# vim /etc/containerd/config.toml 文件
```

将 config_path = ""修改成如下目录：config_path = "/etc/containerd/certs.d"

```
# mkdir /etc/containerd/certs.d/docker.io/ -p
# vim /etc/containerd/certs.d/docker.io/hosts.toml
[host."https://vh3bm52y.mirror.aliyuncs.com",host."https://registry.docker-cn.com"]
  capabilities = ["pull"]
# systemctl restart containerd
```

1.2.6 安装 docker 服务

备注：docker 也要安装，docker 跟 containerd 不冲突，安装 docker 是为了能基于 dockerfile 构建镜像

```
# yum install docker-ce -y
# systemctl enable docker --now
```

1.2.7 配置 docker 镜像加速器

```
# vim /etc/docker/daemon.json     //配置 docker 镜像加速器
{
  "registry-mirrors":["https://vh3bm52y.mirror.aliyuncs.com","https://registry.docker-cn.com","https://docker.mirrors.ustc.edu.cn","https://dockerhub.azk8s.cn","http://hub-mirror.c.163.com"]
}
# systemctl daemon-reload
# systemctl restart docker
```

1.3 部署 k8s 集群

注：未标明具体节点的操作需要在三个节点上都执行。

1.3.1 配置安装 k8s 组件需要的阿里云的 repo 源

```
# vim /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86_64/
enabled=1
gpgcheck=0
```

1.3.2 安装 k8s 初始化工具

```
# yum install -y kubelet-1.25.0 kubeadm-1.25.0 kubectl-1.25.0
# systemctl enable kubelet
```

Kubeadm：kubeadm 是一个工具，用来初始化 k8s 集群的

kubelet：安装在集群所有节点上，用于启动 Pod 的

kubectl：通过 kubectl 可以部署和管理应用，查看各种资源，创建 删除和更新各种组件

1.3.3 设置容器运行时的 endpoing

```
# crictl config runtime-endpoint /run/containerd/containerd.sock
```

1.3.4 使用 kubeadm 初始化 k8s 集群

```
[root@master1 ~]# kubeadm config print init-defaults > kubeadm.yaml
```

根据我们自己的需求修改配置，比如修改 imageRepository 的值，kube-proxy 的模式为 ipvs，需要注意的是由于我们使用的 containerd 作为运行时，所以在初始化节点的时候需要指定 cgroupDriver 为 systemd。

```
# vim kubeadm.yaml
apiVersion: kubeadm.k8s.io/v1beta3
...
kind: InitConfiguration
localAPIEndpoint:
  advertiseAddress: 192.168.40.180    #控制节点的 ip
  bindPort: 6443
nodeRegistration:
  criSocket: unix:///run/containerd/containerd.sock    #指定 containerd 容器运行时的 endpoint
  imagePullPolicy: IfNotPresent
  name:  master1  #控制节点主机名
  taints: null
---
apiVersion: kubeadm.k8s.io/v1beta3
certificatesDir: /etc/kubernetes/pki
clusterName: kubernetes
controllerManager: {}
dns: {}
etcd:
  local:
    dataDir: /var/lib/etcd
imageRepository: registry.aliyuncs.com/google_containers     #指定从阿里云仓库拉取镜像
kind: ClusterConfiguration
kubernetesVersion: 1.25.0    #k8s 版本
networking:
  dnsDomain: cluster.local
  podSubnet: 10.244.0.0/16     #指定 pod 网段
  serviceSubnet: 10.96.0.0/12 #指定 Service 网段
scheduler: {}
#在文件最后，插入以下内容，（复制时，要带着---）
---
apiVersion: kubeproxy.config.k8s.io/v1alpha1
kind: KubeProxyConfiguration
mode: ipvs
---
```

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
cgroupDriver: systemd
```

1.3.5 kubeadm 初始化 k8s 集群

把初始化 k8s 集群需要的离线镜像包上传到所有节点，手动解压。（这一步可以不做，可以自动拉取镜像。）

```
# ctr -n=k8s.io images import k8s_1.25.0.tar.gz
```

1.3.6 基于 kubeadm.yaml 文件初始化 k8s

```
[root@master1~]# kubeadm init --config=kubeadm.yaml --ignore-preflight-errors=SystemVerification
```

1.3.7 配置 kubectl 的配置文件

配置 kubectl 的配置文件 config，相当于对 kubectl 进行授权，这样 kubectl 命令可以使用此证书对 k8s 集群进行管理

```
[root@master1 ~]# mkdir -p $HOME/.kube
[root@master1 ~]# cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
[root@master1 ~]# chown $(id -u):$(id -g) $HOME/.kube/config
[root@master1 ~]# kubectl get nodes
master1          NotReady    control-plane,master    60s    v1.20.6
```

此时集群状态还是 NotReady 状态，因为没有安装网络插件。

1.3.8 扩容 k8s 集群-添加 node 节点

在 master1 上查看加入节点的命令：

```
[root@master1 ~]# kubeadm token create --print-join-command
kubeadm join 192.168.40.199:16443 --token y23a82.hurmcpzedblv34q8 --discovery-token-ca-cert-hash
sha256:1ba1b274090feecfef58eddc2a6f45590299c1d0624618f1f429b18a064cb728
```

把 node1 和 node2 加入 k8s 集群：

```
[root@node1 ~]# kubeadm token create --print-join-command
kubeadm join 192.168.40.199:16443 --token y23a82.hurmcpzedblv34q8 --discovery-token-ca-cert-hash
sha256:1ba1b274090feecfef58eddc2a6f45590299c1d0624618f1f429b18a064cb728 --ignore-preflight-
errors=SystemVerification
[root@node2 ~]# kubeadm token create --print-join-command
kubeadm join 192.168.40.199:16443 --token y23a82.hurmcpzedblv34q8 --discovery-token-ca-cert-hash
sha256:1ba1b274090feecfef58eddc2a6f45590299c1d0624618f1f429b18a064cb728 --ignore-preflight-
errors=SystemVerification
```

```
[root@master1 ~]# kubectl get nodes    //在 master1 上查看集群节点状况
NAME             STATUS     ROLES                  AGE     VERSION
master1          NotReady   control-plane,master   53m     v1.25.0
node1            NotReady   <none>                 58s     v1.25.0
node2            NotReady   <none>                 59s     v1.25.0
```

注意：上面状态都是 NotReady 状态，说明没有安装网络插件。

可以看到 node1 和 node2 的 ROLES 角色为空，<none>就表示这个节点是工作节点。

可以把 node1 和 node2 的 ROLES 变成 work，按照如下方法：

```
[root@master1 ~]# kubectl label node node1 node-role.kubernetes.io/worker=worker
[root@master1 ~]# kubectl label node node2 node-role.kubernetes.io/worker=worker
[root@master1 ~]# kubectl get pods -n kube-system
```

```
NAME                                       READY   STATUS    RESTARTS   AGE
calico-kube-controllers-6744f6b6d5-nck2c   1/1     Running   0          5d21h
calico-node-58tzf                          1/1     Running   0          5d21h
calico-node-gjfb2                          1/1     Running   0          5d21h
calico-node-zvthj                          1/1     Running   0          5d21h
......
```

1.3.9 安装 kubernetes 网络组件-Calico

上传 calico.yaml 到 master1 上，使用 yaml 文件安装 calico 网络插件（这里需要等几分钟才能 ready）。

```
[root@master1 ~]# kubectl apply -f calico.yaml
```

注：在线下载配置文件地址是：https://docs.projectcalico.org/manifests/calico.yaml

```
[root@master1 ~]# kubectl get pods -n kube-system
[root@master1 ~]# kubectl get nodes
NAME      STATUS   ROLES           AGE     VERSION
master1   Ready    control-plane   5d22h   v1.25.0
node1     Ready    work            5d22h   v1.25.0
node2     Ready    work            5d21h   v1.25.0
```

1.3.10 测试在 k8s 创建 pod 是否可以正常访问网络

```
[root@master1 ~]# kubectl run busybox --image busybox:1.28 --image-pull-policy=IfNotPresent --restart=Never --rm -it busybox -- sh
/ # ping www.baidu.com
PING www.baidu.com (39.156.66.18): 56 data bytes
64 bytes from 39.156.66.18: seq=0 ttl=127 time=39.3 ms
```

通过上面可以看到能访问网络，说明 calico 网络插件已经被正常安装了。

1.3.11 测试 coredns 是否正常

```
[root@master1 ~]# kubectl run busybox --image busybox:1.28 --restart=Never --rm -it busybox -- sh
/ # nslookup kubernetes.default.svc.cluster.local
Server:    10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local
```

1.3.12 kubeadm 初始化 k8s 证书过期解决方案

查看证书有效时间：

```
# openssl x509 -in /etc/kubernetes/pki/ca.crt -noout -text  |grep Not
# openssl x509 -in /etc/kubernetes/pki/apiserver.crt -noout -text |grep Not
```

延长证书过期时间：

1.把 update-kubeadm-cert.sh 文件上传到 master1 节点

2.给 update-kubeadm-cert.sh 证书授权可执行权限

```
# chmod +x update-kubeadm-cert.sh
```

3.执行下面命令，修改证书过期时间，把时间延长到 10 年

```
# ./update-kubeadm-cert.sh all
```

4.在 master1 节点查询 pod 是否正常,能查询出数据说明证书签发完成

```
# kubectl get pods -n kube-system
```

5.再次查看证书有效期，可以看到会延长到 10 年

```
# openssl x509 -in /etc/kubernetes/pki/ca.crt -noout -text  |grep Not
# openssl x509 -in /etc/kubernetes/pki/apiserver.crt -noout -text  |grep Not
```

1.3.13 配置其他节点可执行 kubectl 命令

备注：以 node1 节点为例

[root@node1 ~]# mkdir /root/.kube　　//在需要执行 kubectl 命令的节点上创建.kube 目录

[root@master1 ~]# scp /root/.kube/config node1:/root/.kube/　　//从 master1 上拷贝 config 文件到 node1

2 k8s 核心资源 Pod 介绍

2.1 Pod 是什么

K8s 官方文档：https://kubernetes.io/

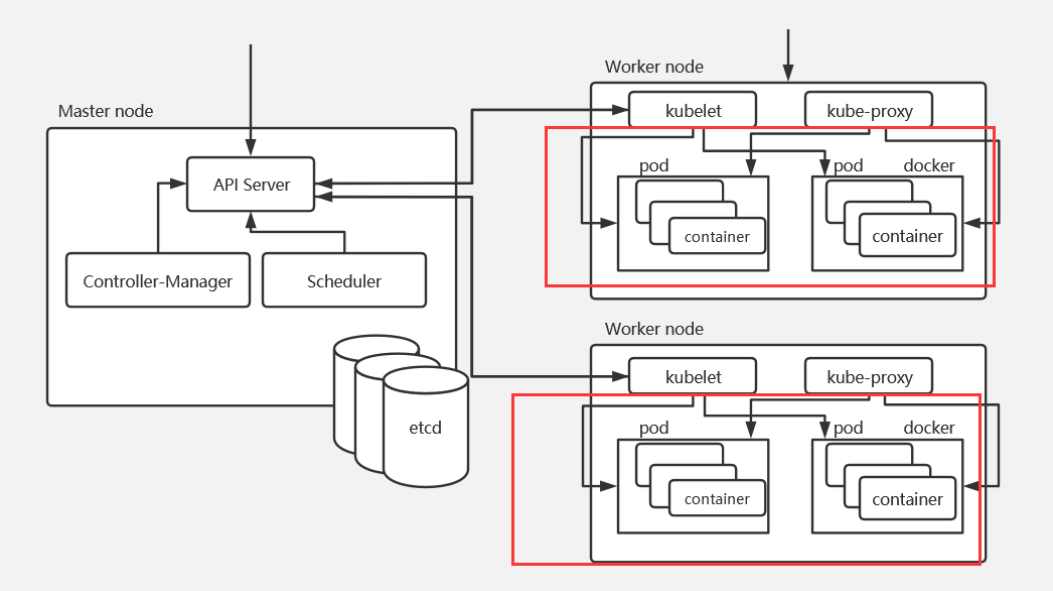
K8s 中文官方文档：https://kubernetes.io/zh/

K8s Github 地址：https://github.com/kubernetes/kubernetes

Pod 资源对应的官方文档：https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/

Pod 是 Kubernetes 中的最小调度单元，k8s 是通过定义一个 Pod 的资源，然后在 Pod 里面运行容器，容器需要指定一个镜像，这样就可以用来运行具体的服务。一个 Pod 封装一个容器（也可以封装多个容器），Pod 里的容器共享存储、网络等。也就是说，应该把整个 pod 看作虚拟机，然后每个容器相当于运行在虚拟机的进程。

Pod 是需要调度到 k8s 集群的工作节点来运行的，具体调度到哪个节点，是根据 scheduler 调度器实现的。



2.1.1 Pod 如何管理多个容器?

Pod 中可以同时运行多个容器。同一个 Pod 中的容器会自动的分配到同一个 node 上。同一个 Pod 中的容器共享资源、网络环境，它们总是被同时调度，在一个 Pod 中同时运行多个容器是一种比较高级的用法，只有当你的容器需要紧密配合协作的时候才考虑用这种模式。例如：你有一个容器作为 web 服务器运行，需要用到共享的 volume，有另一个“sidecar”容器来从远端获取资源更新这些文件。

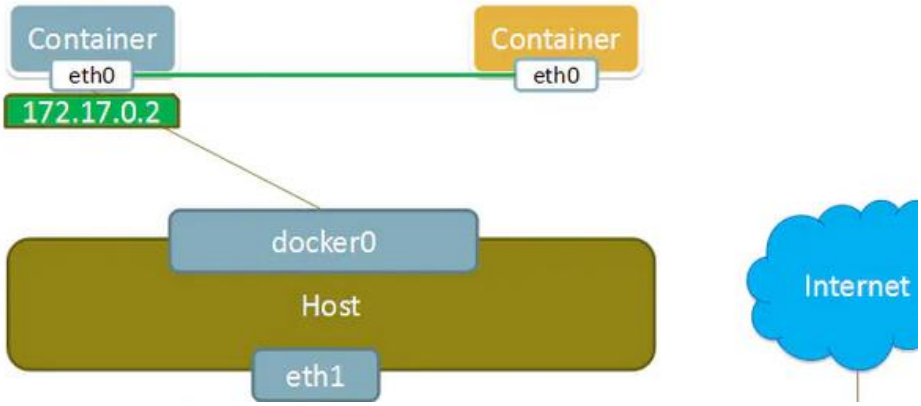
一些 Pod 有 init 容器和应用容器。 在应用程序容器启动之前，运行初始化容器。

2.1.2 Pod 网络

Pod 是有 IP 地址的，假如 pod 不是共享物理机 ip，由网络插件（calico、flannel、weave）划分的 ip，每个 pod 都被分配唯一的 IP 地址

Docker 容器互联的方式：

创建新容器的时候，通过—net container 参数，指定其和已经存在的某个容器共享一个 Network Namespace。如下图所示，右方黄色新创建的 container，其网卡共享左边容器。因此就不会拥有自己独立的 IP，而是共享左边容器的 IP 172.17.0.2，端口范围等网络资源。



和已经存在的 none 容器共享网络：

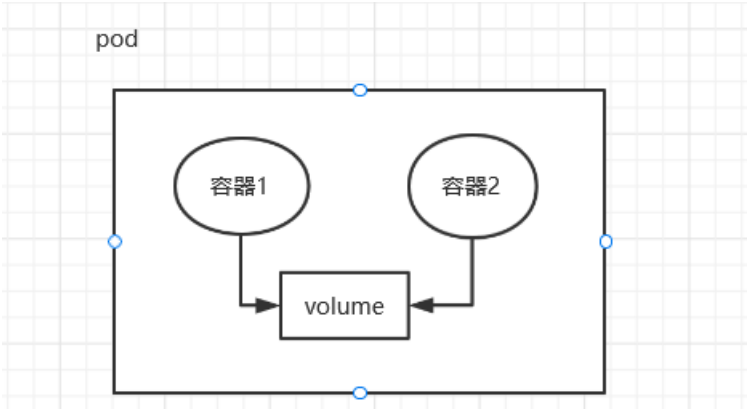
```
# docker run --name container2 --net=container:none -it --privileged=true centos
```

Kubernetes 中容器共享的方式：

在 k8s 中，启动 Pod 时，会先启动一个 pause 的容器，然后将后续的所有容器都 link 到这个 pause 的容器，以实现网络共享。

2.1.3 Pod 存储

创建 Pod 的时候可以指定挂载的存储卷。POD 中的所有容器都可以访问共享卷，允许这些容器共享数据。Pod 只要挂载持久化数据卷，Pod 重启之后数据还是会存在的。



2.2 之前学习过容器，为什么还需要 Pod

1、Pod 是由一组紧耦合的容器组成的容器组，当然目前最流行的就是 Docker、containerd、podman 容器，Pod 就可以作为 1 或者多个容器的载体。

2、Pod 中的所用容器会被一致调度、同节点部署，并且在一个“共享环境”中运行。Pod 想成一个车：车里面好多座位，每个座位都坐不同的人，每个座位想成是一个容器，这里的“共享环境”包括以下几点：

1）所有容器共享一个 IP 地址和端口空间，意味着容器之间可以通过 localhost 高效访问，不能有端口冲突。

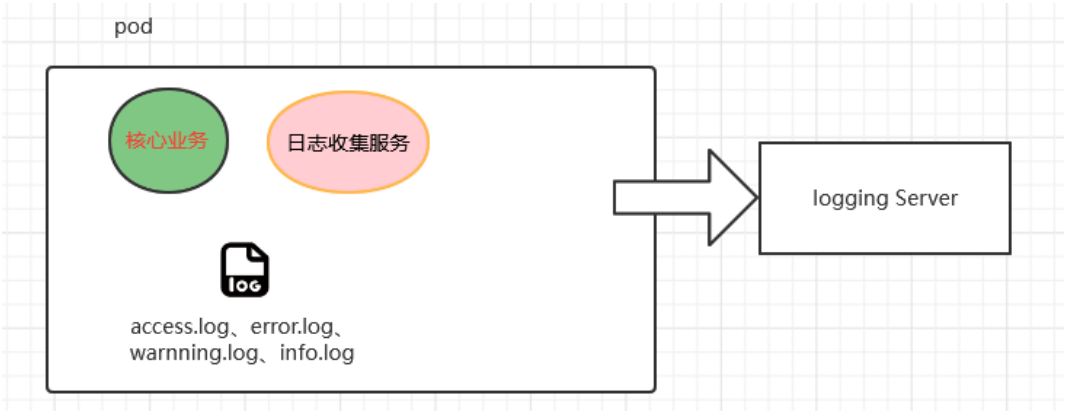
2）允许容器之间共享存储卷，通过文件系统交互信息。

3、有些容器需要紧密联系，需要一起工作。Pod 提供了比容器更高层次的抽象， Pod 中的所有容器使用同一个网络的 namespace，即相同的 IP 地址和 Port 空间。它们可以直接用 localhost 通信。同样的，这些容器可以共享存储，当 K8s 挂载 Volume 到 Pod 上，本质上是将 volume 挂载到 Pod 中的每一个容器里。

2.2.1 代码自动发版更新

假如生产环境部署了一个 go 的应用，而且部署了几百个节点，希望这个应用可以定时的同步最新的代码，以便自动升级线上环境。这时，我们不希望改动原来的 go 应用，可以开发一个 Git 代码仓库的自动同步服务，然后通过 Pod 的方式进行编排，并共享代码目录，就可以达到更新 java 应用代码的效果。

2.2.2 收集业务日志



某服务模块已经实现了一些核心的业务逻辑，并且稳定运行了一段时间，日志记录在了某个目录下，按照不同级别分别为 error.log、access.log、warning.log、info.log，现在希望收集这些日志并发送到统一的日志处理服务器上。

这时我们可以修改原来的服务模块，在其中添加日志收集、发送的服务，但这样可能会影响原来服务的配置、部署方式，从而带来不必要的问题和成本，也会增加业务逻辑和基础服务的藕合度。

如果使用 Pod 的方式，通过简单的编排，既可以保持原有服务逻辑、部署方式不变，又可以增加新的日志收集服务。

而且如果我们对所有服务的日志生成有一个统一的标准，或者仅对日志收集服务稍加修改，就可以将日志收集服务和其他服务进行 Pod 编排，提供统一、标准的日志收集方式。

这里的“核心业务服务”、“日志收集服务”分别是一个镜像，运行在隔离的容器环境中。

2.3 Pod 工作方式

在 K8s 中，所有的资源都可以使用一个 yaml 文件来创建，创建 Pod 也可以使用 yaml 配置文件。或者使用 kubectl run 在命令行创建 Pod（不常用）。

2.3.1 自主式 Pod

所谓的自主式 Pod，就是直接定义一个 Pod 资源，如下：

```
[root@master1 ~]# vim tomcat.yaml
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
  namespace: default
  labels:
    app: myapp
    env: dev
spec:
  containers:
  - name: tomcat-pod-java
    ports:
    - containerPort: 8080
    image: tomcat:8.5-jre8-alpine
    imagePullPolicy: IfNotPresent
[root@master1 ~]# kubectl apply -f tomcat.yaml  //更新资源清单文件
[root@master1 ~]# kubectl get pods -o wide -l app=myapp   //查看 pod 是否创建成功
demo-pod   1/1     Running   0          41s   10.244.104.7   node2
[root@master1 ~]# kubectl delete pods demo-pod    //删除 pod
```

[root@master1 ~]# kubectl delete -f tomcat.yaml //删除 pod

通过上面可以看到，如果直接定义一个 Pod 资源，那 Pod 被删除，就彻底被删除了，不会再创建一个新的 Pod，这在生产环境还是具有非常大风险的，所以今后我们接触的 Pod，都是控制器管理的。

2.3.2 控制器管理的 Pod

常见的管理 Pod 的控制器：Replicaset、Deployment、Job、CronJob、Daemonset、Statefulset。

控制器管理的 Pod 可以确保 Pod 始终维持在指定的副本数运行。

```
[root@master1 ~]# vim nginx-deploy.yaml  //创建一个资源清单文件
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-test
  labels:
    app: nginx-deploy
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx:latest
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
[root@master1 ~]# kubectl apply -f nginx-deploy.yaml      //更新资源清单文件
[root@master1 ~]# kubectl get deploy -l app=nginx-deploy  //查看 Deployment
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
nginx-test       2/2        2            2           16s
[root@master1 ~]# kubectl get rs -l app=nginx  //查看 Replicaset
NAME                   DESIRED   CURRENT   READY   AGE
nginx-test-75c685fdb7   2         2         2       71s
[root@master1 ~]# kubectl get pods -o wide -l app=nginx  //查看 pod
NAME                           READY   STATUS        IP
nginx-test-75c685fdb7-6d4lx    1/1     Running       10.244.102.69
nginx-test-75c685fdb7-9s95h    1/1     Running       10.244.102.68
```

删除 nginx-test-75c685fdb7-9s95h 这个 pod 后发现重新创建一个新的 pod 是 nginx-test-75c685fdb7-pr8gh

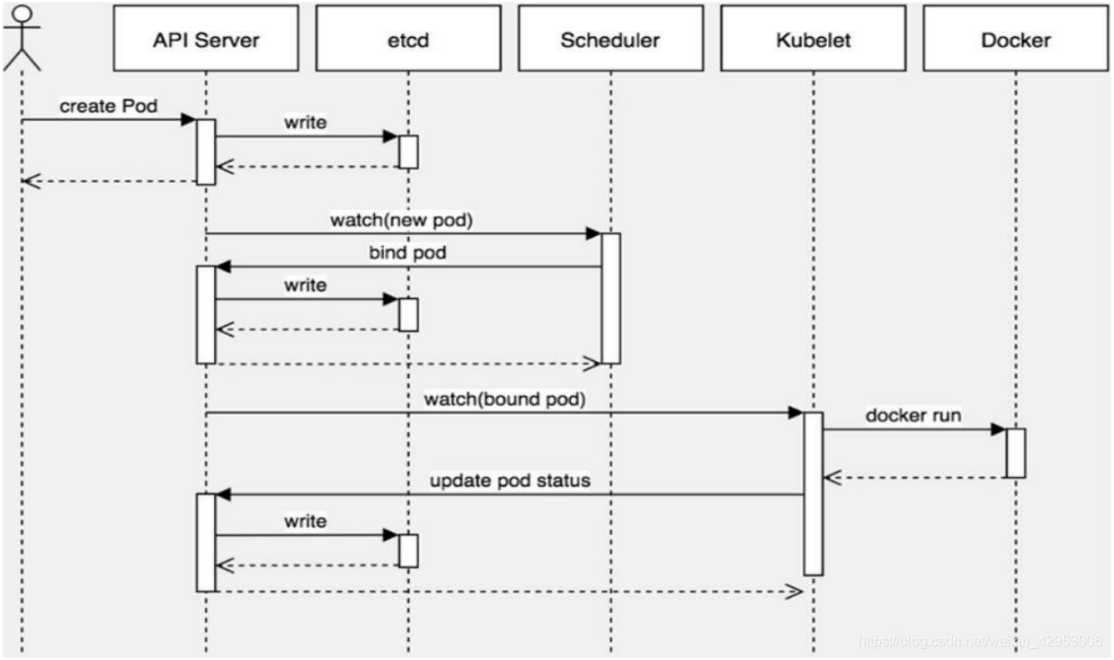
```
[root@master1 ~]# kubectl delete pods nginx-test-75c685fdb7-9s95h
[root@master1 ~]# kubectl get pods -o wide -l app=nginx
NAME                           READY   STATUS              IP
```

```
nginx-test-75c685fdb7-6d4lx     1/1     Running             10.244.102.69
nginx-test-75c685fdb7-pr8gh     1/1     Running             10.244.102.70
```

通过上面可以发现通过 deployment 管理的 pod，可以确保 pod 始终维持在指定副本数量。

2.3.3 如何基于 Pod 运行应用

创建 pod 流程：



kubectl apply -f nginx-deploy.yaml->找到 config 文件，基于 config 文件指定的用户访问指定的集群，这样就找到了 apiserver。

1.通过 kubectl 命令向 apiserver 提交创建 pod 的请求，apiserver 接收到 pod 创建请求后，会将 pod 的属性信息(metadata)写入 etcd。

2.apiserver 触发 watch 机制准备创建 pod，信息转发给调度器 scheduler，调度器使用调度算法选择 node，调度器将 node 信息给 apiserver，apiserver 将绑定的 node 信息写入 etcd。

3.apiserver 又通过 watch 机制，调用 kubelet，指定 pod 信息，调用容器运行时创建并启动 pod 内的容器。

4.创建完成之后反馈给 kubelet，kubelet 又将 pod 的状态信息给 apiserver,apiserver 又将 pod 的状态信息写入 etcd。

2.3.4 通过资源清单文件创建第一个 Pod

```
[root@master1 ~]# vim pod-first.yaml
apiVersion: v1
kind: Pod
metadata:
  name: tomcat-test
  namespace: default
  labels:
    app: tomcat
spec:
  containers:
  - name: tomcat-java
    ports:
    - containerPort: 8080
    image: tomcat:8.5-jre8-alpine
```

```
    imagePullPolicy: IfNotPresent
[root@master1 ~]# kubectl apply -f pod-first.yaml   //更新资源清单文件
[root@master1 ~]# kubectl get pods -l app=tomcat    //查看 pod 是否创建成功
[root@master1 ~]# kubectl get pods -owide    //查看 pod 的 ip 和 pod 调度到哪个节点上
[root@master1 ~]# kubectl logs tomcat-test   //查看 pod 日志
[root@master1 ~]# kubectl exec -it -c tomcat-java tomcat-test -- /bin/bash  //进入到刚才创建的 pod
[root@master1 ~]# kubectl describe pods tomcat-test  //查看 pod 详细信息
[root@master1 ~]# kubectl get pods --show-labels     //查看 pod 具有哪些标签
[root@master1 ~]# kubectl delete pods tomcat-test    //删除 pod
[root@master1 ~]# kubectl delete -f pod-first.yaml   //删除 pod
```

2.3.5 通过 kubectl run 创建 Pod

```
[root@master1 ~]# kubectl run tomcat --image=tomcat:8.5-jre8-alpine --image-pull-policy='IfNotPresent' --port=8080
[root@master1 ~]# kubectl --help  #查看 kubectl 的帮助命令
```

2.4 Pod 资源清单字段解读

通过 kubectl explain 查看定义 Pod 资源包含哪些字段。

```
[root@master1 ~]# kubectl explain pod
FIELDS:
   apiVersion <string>  [APIVersion 定义了对象,代表了一个版本。]
   kind   <string>  [Kind 是字符串类型的值，代表了要创建的资源。]
   metadata    <Object>  [metadata 是对象，定义元数据属性信息的。]
   spec   <Object>  [spec 制定了定义 Pod 的规格，里面包含容器的信息。]
   status <Object>  [status 表示状态，这个不可以修改，定义 pod 的时候也不需要定义这个字段。]
[root@master1 ~]# kubectl explain pod.metadata
RESOURCE: metadata <Object>  # metadata 是对象<Object>，下面可以有多个字段
FIELDS:
   annotations <map[string]string>    #annotations 是注解，map 类型表示对应的值是 key-value 键值对，<string,string>表示 key 和 value 都是 String 类型的
  "metadata": {
    "annotations": {
      "key1" : "value1",
      "key2" : "value2"
    }
  }
```

用 Annotation 来记录的信息包括：

build 信息、release 信息、Docker 镜像信息等，例如时间戳、release id 号、镜像 hash 值、docker registry 地址等；

日志库、监控库、分析库等资源库的地址信息；

程序调试工具信息，例如工具名称、版本号等；

团队的联系信息，例如电话号码、负责人名称、网址等。

```
   clusterName <string>   #对象所属群集的名称。这是用来区分不同集群中具有相同名称和命名空间的资源。此字段现在未设置在任何位置，apiserver 将忽略它，如果设置了就使用设置的值。
   creationTimestamp    <string>
   deletionGracePeriodSeconds    <integer>
```

```
   deletionTimestamp    <string>
   finalizers <[]string>
   generateName    <string>
   generation <integer>
   labels <map[string]string> #labels 是标签，labels 是 map 类型，map 类型表示对应的值是 key-value 键值对，<string,string>表示 key 和 value 都是 String 类型的
   managedFields    <[]Object>
   name    <string>    #创建的资源的名字
   namespace    <string>  # namespaces 划分了一个空间，在同一个 namesace 下的资源名字是唯一的，默认的名称空间是 default。
   ownerReferences <[]Object>
   resourceVersion <string>
   selfLink    <string>
   uid    <string>
[root@master1 ~]# kubectl explain pod.spec
FIELDS:
   activeDeadlineSeconds <integer> #表示 Pod 可以运行的最长时间，达到设置的值后，Pod 会自动停止。
   affinity    <Object>  #定义亲和性的
   automountServiceAccountToken  <boolean>
   containers <[]Object> -required-  #containers 是对象列表，用来定义容器的，是必须字段。对象列表 表示下面有很多对象，对象列表下面的内容用 - 连接。
   dnsConfig  <Object>
   dnsPolicy  <string>
   enableServiceLinks   <boolean>
   ephemeralContainers  <[]Object>
   hostAliases <[]Object>
   hostIPC    <boolean>
   hostNetwork <boolean>
   hostPID    <boolean>
   hostname   <string>
   imagePullSecrets <[]Object>
   initContainers  <[]Object>
   nodeName   <string>
   nodeSelector    <map[string]string>
   overhead   <map[string]string>
   preemptionPolicy <string>
   priority   <integer>
   priorityClassName    <string>
   readinessGates  <[]Object>
   restartPolicy   <string>
   runtimeClassName <string>
   schedulerName   <string>
   securityContext <Object>
   serviceAccount  <string>
```

```
   serviceAccountName   <string>
   setHostnameAsFQDN    <boolean>
   shareProcessNamespace    <boolean>
   subdomain   <string>
   terminationGracePeriodSeconds <integer>
   tolerations <[]Object>
   topologySpreadConstraints <[]Object>
   volumes    <[]Object>
[root@master1 ~]# kubectl explain pod.spec.containers
#container 是定义在 pod 里面的，一个 pod 至少要有一个容器。
FIELDS:
   args   <[]string>
   command    <[]string>
   env    <[]Object>
   envFrom    <[]Object>
   image  <string>  #image 是用来指定容器需要的镜像的
   imagePullPolicy <string>  #镜像拉取策略，pod 是要调度到 node 节点的，那 pod 启动需要镜像，可以根据这个字段设置镜像拉取策略，支持如下三种：
     Always：不管本地是否存在镜像，都要重新拉取镜像
     Never： 从不拉取镜像
     IfNotPresent：如果本地存在，使用本地的镜像，本地不存在，从官方拉取镜像
   lifecycle   <Object>
   livenessProbe    <Object>
   name   <string> -required-  #name 是必须字段，用来指定容器名字的
   ports  <[]Object>  #port 是端口，属于对象列表
   readinessProbe   <Object>
   resources   <Object>
   securityContext <Object>
   startupProbe    <Object>
   stdin  <boolean>
   stdinOnce   <boolean>
   terminationMessagePath    <string>
   terminationMessagePolicy <string>
   tty    <boolean>
   volumeDevices    <[]Object>
   volumeMounts    <[]Object>
   workingDir <string>
[root@master1 ~]# kubectl explain pod.spec.containers.ports
FIELDS:
   containerPort  <integer> -required-  #containerPort 是必须字段， pod 中的容器需要暴露的端口。
   hostIP  <string>  #将容器中的服务暴露到宿主机的端口上时，可以指定绑定的宿主机 IP。
   hostPort  <integer>  #容器中的服务在宿主机上映射的端口
   name  <string>  #端口的名字
   protocol  <string>
```

3 命名空间

3.1 什么是命名空间?

Kubernetes 支持多个虚拟集群，它们底层依赖于同一个物理集群。 这些虚拟集群被称为命名空间。

命名空间 namespace 是 k8s 集群级别的资源，可以给不同的用户、租户、环境或项目创建对应的命名空间，例如： test、devlopment、production 环境分别创建各自的命名空间。

3.2 namespace 应用场景

命名空间适用于存在很多跨多个团队或项目的用户的场景。对于只有几到几十个用户的集群，根本不需要创建或考虑命名空间。

3.3 namespacs 使用案例

3.3.1 创建名称空间

```
[root@master1 ~]# kubectl create ns test  //创建一个 test 命名空间
```

3.3.2 namespace 资源限额

```
[root@master1 ~]# vim namespace-quota.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-quota
  namespace: test
spec:
  hard:
    requests.cpu: "2"
    requests.memory: 2Gi
    limits.cpu: "4"
    limits.memory: 4Gi
[root@master1 ~]# kubectl apply -f namespace-quota.yaml
```

每个容器必须设置内存请求（memory request），内存限额（memory limit），cpu 请求（cpu request）和 cpu 限额（cpu limit）。

所有容器的内存请求总额不得超过 2GiB。

所有容器的内存限额总额不得超过 4GiB。

所有容器的 CPU 请求总额不得超过 2CPU。

所有容器的 CPU 限额总额不得超过 4CPU。

ResouceQuota 对象是在我们的名称空间中创建的，并准备好控制该名称空间中的所有容器的总请求和限制。

```
[root@master1 ~]# vim pod-test.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-test
  namespace: test
  labels:
    app: tomcat-pod-test
```

```
spec:
  containers:
  - name: tomcat-test
    ports:
    - containerPort: 8080
    image: tomcat:8.5-jre8-alpine
    imagePullPolicy: IfNotPresent
    resources:
      requests:
        memory: "100Mi"
        cpu: "500m"
      limits:
        memory: "2Gi"
        cpu: "2"
[root@master1 ~]# kubectl apply -f pod-test.yaml
```

4 标签

4.1 什么是标签?

标签其实就一对 key/value，被关联到对象上，比如 Pod,标签的使用我们倾向于能够表示对象的特殊特点，就是一眼就看出了这个 Pod 是干什么的，标签可以用来划分特定的对象（比如版本，服务类型等），标签可以在创建一个对象的时候直接定义，也可以在后期随时修改，每一个对象可以拥有多个标签，但是，key 值必须是唯一的。创建标签之后也可以方便我们对资源进行分组管理。如果对 pod 打标签，之后就可以使用标签来查看、删除指定的 pod。

在 k8s 中，大部分资源都可以打标签。

4.2 给 pod 资源打标签

```
[root@master1 ~]# kubectl apply -f pod-first.yaml
[root@master1 ~]# kubectl label pods tomcat-test release=v1  //对已经存在的 pod 打标签
[root@master1 ~]# kubectl get pods tomcat-test --show-labels  //查看标签是否打成功
NAME          READY   STATUS    RESTARTS   AGE   LABELS
tomcat-test   1/1     Running   1          21h   release=v1, app=tomcat-pod-first
```

4.3 查看资源标签

```
[root@master1 ~]# kubectl get pods --show-labels   //查看默认名称空间下所有 pod 资源的标签
[root@master1 ~]# kubectl get pods tomcat-test --show-labels  //查看默认名称空间下指定 pod 具有的所有标签
[root@master1 ~]# kubectl get pods -l release  //列出默认名称空间下标签 key 是 release 的 pod
[root@master1 ~]# kubectl get pods -l release=v1  //列出默认名称空间标签 key 是 release，值是 v1 的 pod
[root@master1 ~]# kubectl get pods -L release  //列出默认名称空间标签是 release 的 pod 并打印对应的标签值
[root@master1 ~]# kubectl get pods --all-namespaces --show-labels  //查看所有名称空间下的所有 pod 的标签
```

5 node 节点选择器

我们在创建 pod 资源的时候，pod 会根据 schduler 进行调度，那么默认会调度到随机的一个工作节点，如果我们想要 pod

调度到指定节点或者调度到一些具有相同特点的 node 节点，可以使用 pod 中的 nodeName 或者 nodeSelector 字段指定要调度到的 node 节点。

5.1 nodeName

```
[root@master1 ~]# vim pod-node.yaml
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
  namespace: default
  labels:
    app: myapp
    env: dev
spec:
  nodeName: node1
  containers:
  - name: tomcat-pod-java
    ports:
    - containerPort: 8080
    image: tomcat:8.5-jre8-alpine
    imagePullPolicy: IfNotPresent
  - name: busybox
    image: busybox:1.28
    imagePullPolicy: IfNotPresent
    command:
    - "/bin/sh"
    - "-c"
    - "sleep 3600"
[root@master1 ~]# kubectl apply -f pod-node.yaml
[root@master1 ~]# kubectl get pods -o wide     //查看 pod 调度到哪个节点
demo-pod    2/2     Running    0          2m59s    10.244.166.134    node1
```

5.2 nodeSelector

指定 pod 调度到具有哪些标签的 node 节点上。

```
[root@master1 ~]# kubectl label nodes node2 disk=ceph    //给 node 节点打标签，打个具有 disk=ceph 的标签
[root@master1 ~]# vim pod-1.yaml    //定义 pod 的时候指定要调度到具有 disk=ceph 标签的 node 上
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod-1
  namespace: default
  labels:
    app: myapp
    env: dev
```

```
spec:
  nodeSelector:
    disk: ceph
  containers:
  - name: tomcat-pod-java
    ports:
    - containerPort: 8080
    image: tomcat:8.5-jre8-alpine
    imagePullPolicy: IfNotPresent
[root@master1 ~]# kubectl apply -f pod-1.yaml
[root@master1 ~]# kubectl get pods -o wide    //查看 pod 调度到哪个节点
demo-pod-1   1/1     Running   0          68s   10.244.104.15   node2
[root@master1 ~]# kubectl label nodes node2 disk-    //删除 node 节点打的标签
```

同一个 yaml 文件里定义 pod 资源，如果同时定义了 nodeName 和 NodeSelector，那么条件必须都满足才可以，有一个不满足都会调度失败。

6 污点、容忍度、亲和性

6.1 node 节点亲和性

6.1.1 node 节点亲和性调度：nodeAffinity

```
[root@master1 ~]# kubectl explain pods.spec.affinity
KIND: Pod
VERSION: v1
RESOURCE: affinity <Object>
FIELDS:
   nodeAffinity <Object>
   podAffinity  <Object>
   podAntiAffinity <Object>
[root@master1 ~]# kubectl explain pods.spec.affinity.nodeAffinity
FIELDS:
   preferredDuringSchedulingIgnoredDuringExecution  <[]Object>
   requiredDuringSchedulingIgnoredDuringExecution   <Object>
```

prefered 表示有节点尽量满足这个位置定义的亲和性，这不是一个必须的条件，软亲和性

require 表示必须有节点满足这个位置定义的亲和性，这是个硬性条件，硬亲和性

```
[root@master1 ~]# kubectl explain
pods.spec.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution
FIELDS:
   nodeSelectorTerms    <[]Object> -required-
     Required. A list of node selector terms. The terms are ORed.
[root@master1 ~]# kubectl explain
pods.spec.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms
FIELDS:
   matchExpressions<[]Object>
```

matchFields <[]Object>

matchExpressions：匹配表达式的

matchFields：匹配字段的

[root@master1 ~]# **kubectl explain pods.spec.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms.matchFields**

FIELDS:

key <string> -required-

values <[]string>

[root@master1 ~]# **kubectl explain pods.spec.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms.matchExpressions**

FIELDS:

key <string> -required-

operator <string> -required-

Possible enum values:

- `"DoesNotExist"`
- `"Exists"`
- `"Gt"`
- `"In"`
- `"Lt"`
- `"NotIn"`

values <[]string>

key：检查 label

operator：做等值选则还是不等值选则

values：给定值

6.1.2 使用 requiredDuringSchedulingIgnoredDuringExecution 硬亲和性

把 myapp-v1.tar.gz 上传到 node2 和 node1 上，手动解压：

```
[root@node1 ~]# ctr -n=k8s.io images import myapp-v1.tar.gz
[root@node2 ~]# ctr -n=k8s.io images import myapp-v1.tar.gz
[root@master1 ~]# vim pod-nodeaffinity-demo.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-node-affinity-demo
  namespace: default
  labels:
    app: myapp
    tier: frontend
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
```

```
        - matchExpressions:
          - key: zone
            operator: In
            values:
            - foo
            - bar
  containers:
  - name: myapp
    image: docker.io/ikubernetes/myapp:v1
    imagePullPolicy: IfNotPresent
```

我们检查当前节点中有任意一个节点拥有 zone 标签的值是 foo 或者 bar，就可以把 pod 调度到这个 node 节点的 foo 或者 bar 标签上的节点上。

```
[root@master1 ~]# kubectl apply -f pod-nodeaffinity-demo.yaml
[root@master1 ~]# kubectl get pods -o wide | grep pod-node
pod-node-affinity-demo             0/1     Pending     0    node1
```

status 的状态是 pending，上面说明没有完成调度，因为没有一个拥有 zone 的标签的值是 foo 或者 bar，而且使用的是硬亲和性，必须满足条件才能完成调度。给这个 node1 节点打上标签 zone=foo 再查看。

```
[root@master1 ~]# kubectl label nodes node1 zone=foo
[root@master1 ~]# kubectl get pods -o wide
pod-node-affinity-demo    1/1     Running  0    node1
```

6.1.3 使用 preferredDuringSchedulingIgnoredDuringExecution 软亲和性

```
[root@master1 ~]# vim pod-nodeaffinity-demo-2.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-node-affinity-demo-2
  namespace: default
  labels:
    app: myapp
    tier: frontend
spec:
  containers:
  - name: myapp
    image: docker.io/ikubernetes/myapp:v1
    imagePullPolicy: IfNotPresent
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - preference:
          matchExpressions:
          - key: zone1
            operator: In
            values:
            - foo1
```

```
            - bar1
        weight: 10
      - preference:
          matchExpressions:
          - key: zone2
            operator: In
            values:
            - foo2
            - bar2
        weight: 20
[root@master1 ~]# kubectl apply -f pod-nodeaffinity-demo-2.yaml
[root@master1 ~]# kubectl get pods -o wide | grep demo-2
pod-node-affinity-demo-2           1/1     Running     0          node1
```

上面说明软亲和性是可以运行这个 pod 的，尽管没有运行这个 pod 的节点定义的 zone1 标签。

Node 节点亲和性针对的是 pod 和 node 的关系，Pod 调度到 node 节点的时候匹配的条件。

测试 weight 权重：

weight 是相对权重，权重越高，pod 调度的几率越大假如给 node1 和 node2 都打上标签：

```
[root@master1 ~]# kubectl label nodes node1 zone1=foo1
[root@master1 ~]# kubectl label nodes node2 zone2=foo2
preferredDuringSchedulingIgnoredDuringExecution:
      - preference:
          matchExpressions:
          - key: zone1
            operator: In
            values:
            - foo1
            - bar1
        weight: 10
      - preference:
          matchExpressions:
          - key: zone2
            operator: In
            values:
            - foo2
            - bar2
        weight: 20
```

pod 在定义 node 节点亲和性的时候,node1 和 node2 都满足条件,都可以调度 pod,但是 node2 具有的标签是 zone2=foo2，pod 在匹配 zone2=foo2 的权重高，那么 pod 就会优先调度到 node2 上。

```
[root@master1 ~]# kubectl label nodes node1 zone1-
[root@master1 ~]# kubectl label nodes node2 zone2-
[root@master1 ~]# kubectl delete -f pod-nodeaffinity-demo-2.yaml
```

6.2 Pod 节点亲和性

6.2.1 pod 亲和性调度的两种表示形式

podaffinity：pod 和 pod 更倾向腻在一起，把相近的 pod 结合到相近的位置，如同一区域，同一机架，这样的话 pod 和 pod 之间更好通信，比方说有两个机房，这两个机房部署的集群有 1000 台主机，那么我们希望把 nginx 和 tomcat 都部署同一个地方的 node 节点上，可以提高通信效率；

podunaffinity：pod 和 pod 更倾向不腻在一起，如果部署两套程序，那么这两套程序更倾向于反亲和性，这样相互之间不会有影响。

第一个 pod 随机选则一个节点，做为评判后续的 pod 能否到达这个 pod 所在的节点上的运行方式，这就称为 pod 亲和性；我们怎么判定哪些节点是相同位置的，哪些节点是不同位置的；我们在定义 pod 亲和性时需要有一个前提，哪些 pod 在同一个位置，哪些 pod 不在同一个位置，这个位置是怎么定义的，标准是什么？以节点名称为标准，这个节点名称相同的表示是同一个位置，节点名称不相同的表示不是一个位置。

```
[root@master1 ~]# kubectl explain pods.spec.affinity.podAffinity
KIND: Pod
VERSION: v1
RESOURCE: podAffinity <Object>
FIELDS:
   preferredDuringSchedulingIgnoredDuringExecution  <[]Object>
   requiredDuringSchedulingIgnoredDuringExecution   <[]Object>
```

requiredDuringSchedulingIgnoredDuringExecution： 硬亲和性

preferredDuringSchedulingIgnoredDuringExecution：软亲和性

```
[root@master1 ~]# kubectl explain
pods.spec.affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution
KIND:  Pod
VERSION:  v1
RESOURCE: requiredDuringSchedulingIgnoredDuringExecution <[]Object>
DESCRIPTION:
FIELDS:
   labelSelector   <Object>
   namespaces <[]string>
   topologyKey <string> -required-
```

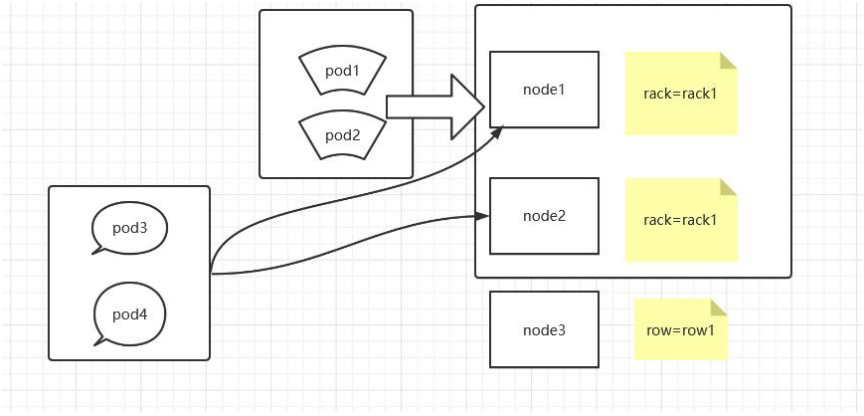
topologyKey：位置拓扑的键，这个是必须字段，是判断节点是不是同一个位置的参数。

rack=rack1

row=row1

使用 rack 的键是同一个位置

使用 row 的键是同一个位置



labelSelector： #我们要判断 pod 跟别的 pod 亲和，跟哪个 pod 亲和，需要靠 labelSelector，通过 labelSelector 选则一组能作为亲和对象的 pod 资源。

namespace： #labelSelector 需要选则一组资源，那么这组资源是在哪个名称空间中呢，通过 namespace 指定，如果不指定 namespaces，那么就是当前创建 pod 的名称空间

[root@master1 ~]# **kubectl explain pods.spec.affinity.podAffinity.requiredDuringSchedulingIgnoredDuringExecution.labelSelector**

KIND: Pod

VERSION: v1

RESOURCE: labelSelector <Object>

FIELDS:

matchExpressions<[]Object>

matchLabels <map[string]string>

6.2.2 pod 节点亲和性

定义两个 pod，第一个 pod 做为基准，第二个 pod 跟着它走。

```
[root@master1 ~]# vim pod-required-affinity-demo-1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-first
  labels:
    app2: myapp2
    tier: frontend
spec:
    containers:
    - name: myapp
      image: ikubernetes/myapp:v1
      imagePullPolicy: IfNotPresent
[root@master1 ~]# kubectl apply -f pod-required-affinity-demo-1.yaml
[root@master1 ~]# vim pod-required-affinity-demo-2.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-second
  labels:
    app: backend
    tier: db
spec:
    containers:
    - name: busybox
      image: busybox:latest
      imagePullPolicy: IfNotPresent
      command: ["sh","-c","sleep 3600"]
    affinity:
      podAffinity:
```

```
        requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
            - key: app2
              operator: In
              values:
              - myapp2
          topologyKey: kubernetes.io/hostname
```

上面表示创建的 pod 必须与拥有 app=myapp 标签的 pod 在一个节点上。

```
[root@master1 ~]# kubectl apply -f pod-required-affinity-demo-2.yaml
[root@master1 ~]# kubectl get pods -o wide 显示如下：
pod-first              running        node2
pod-second             running        node2
[root@master1 ~]# kubectl delete -f pod-required-affinity-demo-1.yaml  //删除 pod
[root@master1 ~]# kubectl delete -f pod-required-affinity-demo-2.yaml  //删除 pod
```

6.2.3 pod 节点反亲和性

定义两个 pod，第一个 pod 做为基准，第二个 pod 跟它调度节点相反。

```
[root@master1 ~]# vim pod-required-anti-affinity-demo-1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-first
  labels:
    app1: myapp1
    tier: frontend
spec:
    containers:
    - name: myapp
      image: ikubernetes/myapp:v1
      imagePullPolicy: IfNotPresent
[root@master1 ~]# kubectl apply -f  pod-required-anti-affinity-demo-1.yaml
[root@master1 ~]# vim pod-required-anti-affinity-demo-2.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-second
  labels:
    app: backend
    tier: db
spec:
  containers:
  - name: busybox
    image: busybox:latest
    imagePullPolicy: IfNotPresent
```

```
    command: ["sh","-c","sleep 3600"]
  affinity:
    podAntiAffinity:
       requiredDuringSchedulingIgnoredDuringExecution:
       - labelSelector:
           matchExpressions:
           - key: app1
             operator: In
             values:
             - myapp1
         topologyKey: kubernetes.io/hostname
[root@master1 ~]# kubectl apply -f pod-required-anti-affinity-demo-2.yaml
[root@master1 ~]# kubectl get pods -o wide   //显示两个 pod 不在一个 node 节点上，这就是 pod 节点反亲和性
pod-first            running        node1
pod-second           running        node2
[root@master1 ~]# kubectl delete -f pod-required-anti-affinity-demo-1.yaml  //删除 pod
[root@master1 ~]# kubectl delete -f pod-required-anti-affinity-demo-2.yaml  //删除 pod
```

6.2.4 topologykey

```
[root@master1 ~]# kubectl label nodes node1 zone=foo
[root@master1 ~]# kubectl label nodes node2 zone=foo
[root@master1 ~]# vim pod-first-required-anti-affinity-demo-1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-first
  labels:
    app3: myapp3
    tier: frontend
spec:
    containers:
    - name: myapp
      image: ikubernetes/myapp:v1
      imagePullPolicy: IfNotPresent
[root@master1 ~]# kubectl apply -f pod-first-required-anti-affinity-demo-1.yaml
[root@master1 ~]# vim pod-second-required-anti-affinity-demo-1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-second
  labels:
    app: backend
    tier: db
spec:
    containers:
```

```
    - name: busybox
      image: busybox:latest
      imagePullPolicy: IfNotPresent
      command: ["sh","-c","sleep 3600"]
    affinity:
      podAntiAffinity:
         requiredDuringSchedulingIgnoredDuringExecution:
         - labelSelector:
             matchExpressions:
             - {key: app3 ,operator: In, values: ["myapp3"]}
           topologyKey: zone
[root@master1 ~]# kubectl apply -f pod-second-required-anti-affinity-demo-1.yaml
[root@master1 ~]# kubectl get pods -o wide
pod-first              running         node1
pod-second             pending         <none>
```

第二个 pod 是 pending，因为两个节点是同一个位置，现在没有不是同一个位置的了，而且我们要求反亲和性，所以就会处于 pending 状态，如果在反亲和性这个位置把 required 改成 preferred，那么也会运行。

```
[root@master1 ~]# kubectl delete -f pod-first-required-anti-affinity-demo-1.yaml
[root@master1 ~]# kubectl delete -f pod-second-required-anti-affinity-demo-1.yaml
[root@master1 ~]# kubectl label nodes node1 zone-
[root@master1 ~]# kubectl label nodes node2 zone-
```

6.3 污点、容忍度

6.3.1 污点及容忍度解释

给了节点选则的主动权，我们给节点打一个污点，不容忍的 pod 就运行不上来，污点就是定义在节点上的键值属性数据，可以定决定拒绝那些 pod。

taints 是键值数据，用在节点上，定义污点

tolerations 是键值数据，用在 pod 上，定义容忍度，能容忍哪些污点

pod 亲和性是 pod 属性，但是污点是节点的属性，污点定义在 k8s 集群的节点上的一个字段。

```
[root@master1 ~]# kubectl explain node.spec.taints
KIND:     Node
VERSION:  v1
RESOURCE: taints <[]Object>
FIELDS:
   effect <string> -required-
   key    <string> -required-
   timeAdded   <string>
   value  <string>
```

taints 的 effect 用来定义对 pod 对象的排斥等级：

NoSchedule：仅影响 pod 调度过程，当 pod 能容忍这个节点污点，就可以调度到当前节点，后来这个节点的污点改了，加了一个新的污点，使得之前调度的 pod 不能容忍了，那这个 pod 会怎么处理，对现存的 pod 对象不产生影响。

NoExecute：既影响调度过程，又影响现存的 pod，如果现存的 pod 不能容忍节点后来加的污点，这个 pod 就会被驱逐。

PreferNoSchedule：最好不，也可以，是 NoSchedule 的柔性版本。

```
[root@master1 ~]# kubectl describe nodes master1
Taints:   node-role.kubernetes.io/control-plane:NoSchedule
```

上面可以看到 master 这个节点的污点是 Noschedule。

所以我们创建的 pod 都不会调度到 master 上，因为我们创建的 pod 没有容忍度。

```
[root@master1 ~]# kubectl describe pods kube-apiserver-master1 -n kube-system
Tolerations:       :NoExecute op=Exists
```

可以看到这个 pod 的容忍度是 NoExecute，则可以调度到 master1 上。

```
[root@master1 ~]# kubectl taint --help  //管理节点污点
```

6.3.2 NoSchedule

把 node2 当成是生产环境专用的，其他 node 是测试的。给 node2 打污点，pod 如果不能容忍就不会调度过来。

```
[root@master1 ~]# kubectl taint node node2 node-type=production:NoSchedule
[root@master1 ~]# vim pod-taint.yaml
apiVersion: v1
kind: Pod
metadata:
  name: taint-pod
  namespace: default
  labels:
    tomcat: tomcat-pod
spec:
  containers:
  - name: taint-pod
    ports:
    - containerPort: 8080
    image: tomcat:8.5-jre8-alpine
    imagePullPolicy: IfNotPresent
[root@master1 ~]# kubectl apply -f pod-taint.yaml
[root@master1 ~]# kubectl get pods -o wide
taint-pod   running    node1
```

可以看到都被调度到 node1 上了，因为 node2 这个节点打了污点，而我们在创建 pod 的时候没有容忍度，所以 node2 上不会有 pod 调度上去的。

6.3.3 NoExecute

```
[root@master1 ~]# kubectl taint node node1 node-type=dev:NoExecute    //给 node1 也打上污点
[root@master1 ~]# kubectl get pods -o wide
taint-pod   termaitering
```

可以看到已经存在的 pod 节点都被撵走了

6.3.4 toleration

```
[root@master1 ~]# vim pod-demo-1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-deploy
  namespace: default
  labels:
```

```
    app: myapp
    release: canary
spec:
  containers:
  - name: myapp
    image: ikubernetes/myapp:v1
    imagePullPolicy: IfNotPresent
    ports:
    - name: http
      containerPort: 80
  tolerations:
  - key: "node-type"
    operator: "Equal"
    value: "production"
    effect: "NoExecute"
    tolerationSeconds: 3600
[root@master1 ~]# kubectl apply -f pod-demo-1.yaml
[root@master1 ~]# kubectl get pods
myapp-deploy   1/1     Pending   0          11s   node2
```

还是显示 pending，因为我们使用的是 equal（等值匹配），所以 key 和 value，effect 必须和 node 节点定义的污点完全匹配才可以，把上面配置 effect: "NoExecute"变成 effect: "NoSchedule"，tolerationSeconds: 3600 这行去掉。

```
[root@master1 ~]# kubectl delete -f pod-demo-1.yaml
[root@master1 ~]# kubectl apply -f pod-demo-1.yaml
[root@master1 ~]# kubectl get pods -o wide
myapp-deploy   1/1     running  0          11s   node2
```

上面就可以调度到 node2 上了，因为在 pod 中定义的容忍度能容忍 node 节点上的污点

再次修改如下部分：

```
      tolerations:
      - key: "node-type"
        operator: "Exists"
        value: ""
        effect: "NoSchedule"
```

只要对应的键是存在的，exists，其值被自动定义成通配符。

```
[root@master1 ~]# kubectl delete -f pod-demo-1.yaml
[root@master1 ~]# kubectl apply -f pod-demo-1.yaml
[root@master1 ~]# kubectl get pods
```

发现还是调度到 node2 上

```
myapp-deploy   1/1     running  0     11s   node2
```

再次修改：

```
      tolerations:
      - key: "node-type"
        operator: "Exists"
```

```
        value: ""
        effect: ""
```

有一个 node-type 的键，不管值是什么，不管是什么效果，都能容忍。

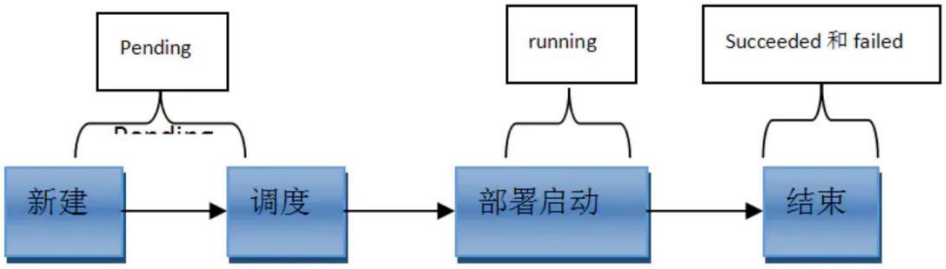
```
[root@master1 ~]# kubectl delete -f pod-demo-1.yaml
[root@master1 ~]# kubectl apply -f pod-demo-1.yaml
[root@master1 ~]# kubectl get pods -o wide 显示如下:
myapp-deploy  running    node1
```

可以看到 node2 和 node1 节点上都有可能有 pod 被调度

```
[root@master1 ~]# kubectl taint nodes node1 node-type:NoExecute-  //删除污点
[root@master1 ~]# kubectl taint nodes node2 node-type-  //删除污点
```

6.4 Pod 常见的状态和重启策略

6.4.1 常见的 pod 状态



第一阶段：

挂起（Pending）：

1、正在创建 Pod 但是 Pod 中的容器还没有全部被创建完成，处于此状态的 Pod 应该检查 Pod 依赖的存储是否有权限挂载、镜像是否可以下载、调度是否正常等。

2、我们在请求创建 pod 时，条件不满足，调度没有完成，没有任何一个节点能满足调度条件，已经创建了 pod 但是没有适合它运行的节点叫做挂起，调度没有完成。

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）：未知状态，所谓 pod 是什么状态是 apiserver 和运行在 pod 节点的 kubelet 进行通信获取状态信息的，如果节点之上的 kubelet 本身出故障，那么 apiserver 就连不上 kubelet，得不到信息了，就会看 Unknown，通常是由于与 pod 所在的 node 节点通信错误。

Error 状态：Pod 启动过程中发生了错误

成功（Succeeded）：Pod 中的所有容器都被成功终止，即 pod 里所有的 containers 均已 terminated。

第二阶段：

Unschedulable：Pod 不能被调度， scheduler 没有匹配到合适的 node 节点。

PodScheduled：pod 正处于调度中，在 scheduler 刚开始调度的时候，还没有将 pod 分配到指定的 node，在筛选出合适的节点后就会更新 etcd 数据，将 pod 分配到指定的 node。

Initialized：所有 pod 中的初始化容器已经完成了。

ImagePullBackOff：Pod 所在的 node 节点下载镜像失败。

Running：Pod 内部的容器已经被创建并且启动。

还有其他状态如下：

Evicted 状态：出现这种情况，多见于系统内存或硬盘资源不足，可 df-h 查看 docker 存储所在目录的资源使用情况，如果百分比大于 85%，就要及时清理下资源，尤其是一些大文件、docker 镜像。

CrashLoopBackOff：容器曾经启动了，但可能又异常退出了。

6.4.2 pod 重启策略

Pod 的重启策略（RestartPolicy）应用于 Pod 内的所有容器，当某个容器异常退出或者健康检查失败时，kubelet 将根据重启策略来进行相应的操作。

Pod 的 spec 中包含一个 restartPolicy 字段，其可能取值包括 Always、OnFailure 和 Never，默认值是 Always。

Always：只要容器异常退出，kubelet 就会自动重启该容器。（这个是默认的重启策略）

OnFailure：当容器终止运行且退出码不为 0 时，由 kubelet 自动重启该容器。

Never：不论容器运行状态如何，kubelet 都不会重启该容器。

6.4.3 测试 Always 重启策略

```
[root@master1 ~]# vim pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
  namespace: default
  labels:
    app: myapp
spec:
  restartPolicy: Always
  containers:
  - name: tomcat-pod-java
    ports:
    - containerPort: 8080
    image: tomcat:8.5-jre8-alpine
    imagePullPolicy: IfNotPresent
[root@master1 ~]# kubectl apply -f pod.yaml
```

1）正常停止容器里的 tomcat 服务

```
[root@master1 ~]# kubectl exec -it demo-pod -- /bin/bash
bash-4.4# /usr/local/tomcat/bin/shutdown.sh
[root@master1 ~]# kubectl get pod  //查看 pod 状态
demo-pod      1/1      Running    1 (5s ago)    3m24s
```

发现正常停止容器里的 tomcat 服务，容器重启了一次，pod 又恢复正常了。

2）非正常停止容器里的 tomcat 服务

```
[root@master1 ~]# kubectl exec -it demo-pod -- /bin/bash
bash-4.4# kill 1
[root@master1 ~]# kubectl get pod
demo-pod      1/1      Running    2 (5s ago)    3m24s
```

上面可以看到容器终止了，并且又重启一次，重启次数增加了一次

6.4.4 测试 never 重启策略

```
[root@master1 ~]# vim pod.yaml
apiVersion: v1
```

```
kind: Pod
metadata:
  name: demo-pod
  namespace: default
  labels:
    app: myapp
spec:
  restartPolicy: Never
  containers:
  - name: tomcat-pod-java
    ports:
    - containerPort: 8080
    image: tomcat:8.5-jre8-alpine
    imagePullPolicy: IfNotPresent
[root@master1 ~]# kubectl apply -f pod.yaml
```

1）正常停止容器里的 tomcat 服务

```
[root@master1 ~]# kubectl exec -it demo-pod -- /bin/bash
bash-4.4# /usr/local/tomcat/bin/shutdown.sh
[root@master1 ~]# kubectl get pod
demo-pod     1/1     Completed  0     3m24s
```

发现正常停止容器里的 tomcat 服务，pod 正常运行，容器没有重启

2）非正常停止容器里的 tomcat 服务

```
[root@master1 ~]# kubectl exec -it tomcat-pod -- /bin/bash
bash-4.4# kill 1
[root@master1 ~]# kubectl get pod
demo-pod     1/1     error     0          3m24s
```

可以看到 pod 状态是 error，并且没有重启，说明重启策略是 never，那么 pod 里容器服务无论如何终止，都不会重启。

6.4.5 测试 OnFailure 重启策略

```
[root@master1 ~]# vim pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
  namespace: default
  labels:
    app: myapp
spec:
  restartPolicy: OnFailure
  containers:
  - name: tomcat-pod-java
    ports:
    - containerPort: 8080
    image: tomcat:8.5-jre8-alpine
    imagePullPolicy: IfNotPresent
```

```
[root@master1 ~]# kubectl apply -f pod.yaml
```

1）正常停止容器里的 tomcat 服务

```
[root@master1 ~]# kubectl exec -it demo-pod -- /bin/bash
bash-4.4# /usr/local/tomcat/bin/shutdown.sh
[root@master1 ~]# kubectl get pod
demo-pod      0/1     complete   0      3m24s
```

正常停止容器里的 tomcat 服务，退出码是 0，pod 里的容器不会重启

2）非正常停止容器里的 tomcat 服务

```
[root@master1 ~]# kubectl exec -it tomcat-pod -- /bin/bash
bash-4.4# kill 1
[root@master1 ~]# kubectl get pod
demo-pod      1/1       running    1          3m24s
```

非正常停止 pod 里的容器，容器退出码不是 0，那就会重启容器

7 Pod 生命周期

7.1 生命周期的几个重要流程

创建主容器（containers）是必须的操作，初始化容器（initContainers），容器启动后钩子，启动探测、存活性探测，就绪性探测，容器停止前钩子。

pod 在整个生命周期的过程中总会处于以下几个状态：

Pending：创建了 pod 资源并存入 etcd 中，但尚未完成调度。

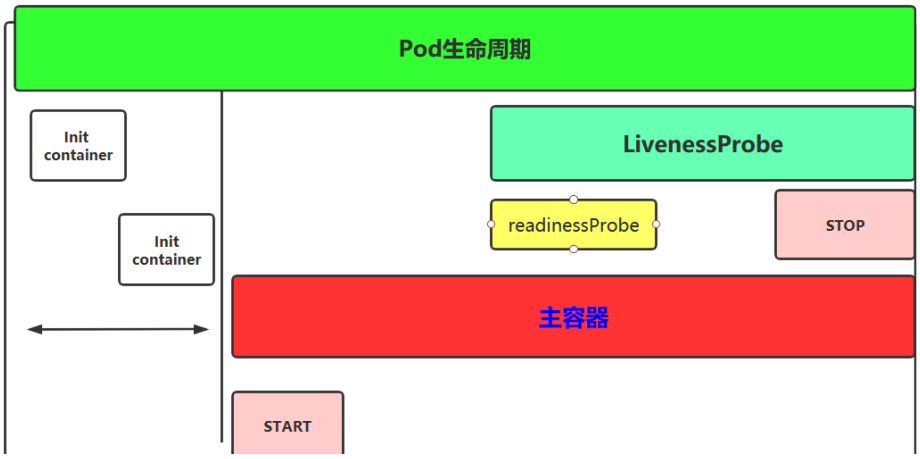
ContainerCreating：Pod 的调度完成，被分配到指定 Node 上。处于容器创建的过程中，通常是在拉取镜像的过程中。

Running：Pod 包含的所有容器都已经成功创建，并且成功运行起来。

Succeeded：Pod 中的所有容器都已经成功终止并且不会被重启

Failed：所有容器都已经终止，但至少有一个容器终止失败，就是说容器返回了非 0 值的退出状态或已经被系统终止。

Unknown：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。



7.2 pod 生命周期的重要行为

1. 在启动任何容器之前，先创建 pause 基础容器，它初始化 Pod 的环境并为后续加入的容器提供共享的名称空间。

2. 初始化容器（initcontainer）：一个 pod 可以拥有任意数量的 init 容器。init 容器是按照顺序以此执行的，并且仅当最后一个 init 容器执行完毕才会去启动主容器。

3.生命周期钩子：pod 允许定义两种类型的生命周期钩子，启动后(post-start)钩子和停止前(pre-stop)钩子，这些生命周期钩子是基于每个容器来指定的，和 init 容器不同的是，init 容器是应用到整个 pod。而这些钩子是针对容器的，是在容器启动后和停止前执行的。

4.容器探测：对 Pod 健康状态诊断。分为三种： Startupprobe、Livenessprobe(存活性探测)， Readinessprobe(就绪性检测)

Startup（启动探测）:探测容器是否正常运行。

Liveness(存活性探测)：判断容器是否处于 runnning 状态，根据重启策略决定是否重启容器。

Readiness(就绪性检测)：判断容器是否准备就绪并对外提供服务，将容器设置为不可用，不接受 service 转发的请求。

三种探针用于 Pod 检测：

ExecAction：在容器中执行一个命令，并根据返回的状态码进行诊断，只有返回 0 为成功。

TCPSocketAction：通过与容器的某 TCP 端口尝试建立连接。

HTTPGetAction：通过向容器 IP 地址的某指定端口的 path 发起 HTTP GET 请求。

5.容器的重启策略：定义是否重启 Pod 对象

Always：但凡 Pod 对象终止就重启，默认设置。

OnFailure：仅在 Pod 出现错误时才重启。

Never：从不。

6.pod 的终止过程，终止过程主要分为如下几个步骤：

1）用户发出删除 pod 命令：kubectl delete pods，kubectl delete -f yaml。

2）Pod 对象随着时间的推移更新，在宽限期（默认情况下 30 秒），pod 被视为“dead”状态。

3）将 pod 标记为“Terminating”状态。

4）第 3 步同时运行，监控到 pod 对象为“Terminating”状态的同时启动 pod 关闭过程。

5）第 3 步同时进行，endpoints 控制器监控到 pod 对象关闭，将 pod 与 service 匹配的 endpoints 列表中删除。

6）如果 pod 中定义了 preStop 钩子处理程序，则 pod 被标记为“Terminating”状态时以同步的方式启动执行；若宽限期结束后，preStop 仍未执行结束，第二步会重新执行并额外获得一个 2 秒的小宽限期。

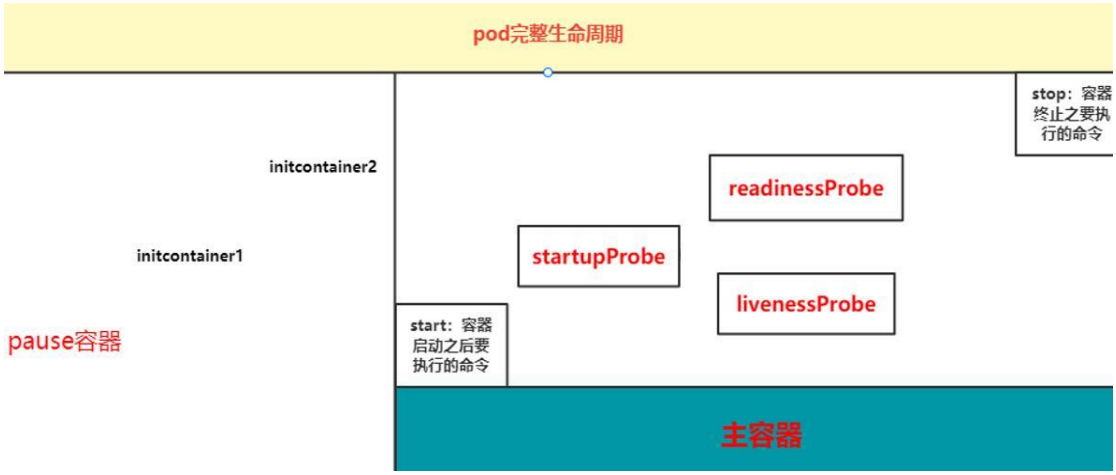
7）Pod 内对象的容器收到 TERM 信号。

8）宽限期结束之后，若存在任何一个运行的进程，pod 会收到 SIGKILL 信号。

9）Kubelet 请求 API Server 将此 Pod 资源宽限期设置为 0 从而完成删除操作。

7.3 Init 容器

spec 字段下有个 initContainers 字段(初始化容器)，Init 容器就是做初始化工作的容器。可以有一个或多个，如果多个按照定义的顺序依次执行，先执行初始化容器 1，再执行初始化容器 2 等，等初始化容器执行完具体操作之后初始化容器就退出了，只有所有的初始化容器执行完后，主容器才启动。



由于一个 Pod 里容器存储卷是共享的，所以 Init Container 里产生的数据可以被主容器使用到，Init Container 可以在多种 K8S 资源里被使用到，如 Deployment、DaemonSet， StatefulSet、Job 等，但都是在 Pod 启动时，在主容器启动前执行，做初始化工作。

初始化容器与主容器区别是:

1、初始化容器不支持 Readinessprobe,因为它们必须在 Pod 就绪之前运行完成。

2、每个 Init 容器必须运行成功,下一个才能够运行。

初始化容器的官方地址：

https://kubernetes.io/docs/concepts/workloads/pods/init-containers/#init-containers-in-use

7.3.1 初始化容器使用案例

```
[root@master1 ~]# vim init.yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    imagePullPolicy: IfNotPresent
    command: ['sh', '-c', "until nslookup myservice.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do echo waiting for myservice;
sleep 2; done"]
  - name: init-mydb
    image: busybox:1.28
    imagePullPolicy: IfNotPresent
    command: ['sh', '-c', "until nslookup mydb.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do echo waiting for mydb; sleep
2; done"]
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
[root@master1 ~]# kubectl apply -f init.yaml
[root@master1 ~]# kubectl get pods
myapp-pod         0/1     Init:0/2   0          2m29s
```

因为初始化容器中定义的域名解析是无法完成的。

修改下如下内容后重试：

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
```

```
    labels:
      app: myapp
  spec:
    initContainers:
    - name: init-myservice
      image: busybox:1.28
      imagePullPolicy: IfNotPresent
      command: ['sh', '-c', "sleep 2"]
    - name: init-mydb
      image: busybox:1.28
      imagePullPolicy: IfNotPresent
      command: ['sh', '-c', "sleep 2"]
    containers:
    - name: myapp-container
      image: busybox:1.28
      imagePullPolicy: IfNotPresent
      command: ['sh', '-c', 'echo containers start running! && sleep 3600']
```

7.3.2 初始化容器生产应用

主容器运行 nginx 服务，初始化容器用来给主容器生成 index.html 文件

```
[root@master1 ~]# vim init-1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: initnginx
spec:
  initContainers:
  - name: install
    image: busybox:1.28
    imagePullPolicy: IfNotPresent
    command:
    - wget
    - "-O"
    - "/work-dir/index.html"
    - "https://www.baidu.com"
    volumeMounts:
    - name: workdir
      mountPath: /work-dir
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 80
    volumeMounts:
```

```
    - name: workdir
      mountPath: /usr/share/nginx/html
  dnsPolicy: Default
  volumes:
  - name: workdir
    emptyDir: {}
[root@master1 ~]# kubectl apply -f init-1.yaml
[root@master1 ~]# kubectl get pods -owide
initnginx   1/1     Running   0          10s   10.244.102.102   node2
[root@master1 ~]# curl 10.244.102.102
[root@master1 ~]# kubectl exec -it initnginx -- ls /usr/share/nginx/html/
index.html
```

7.4 主容器

初始化容器启动之后，开始启动主容器，在主容器启动之后有一个 post start hook（容器启动后钩子）和 pre stop hook（容器结束前钩子），无论启动后还是结束前所做的事我们可以把它放两个钩子，这个钩子就表示用户可以用它来钩住一些命令，非必须选项。

postStart：该钩子在容器被创建后立刻执行，如果该钩子对应的探测执行失败，则该容器会被杀死，并根据该容器的重启策略决定是否要重启该容器，这个钩子不需要传递任何参数。

preStop：该钩子在容器被删除前执行，主要用于释放资源和优雅关闭程序

7.4.1 容器钩子：postStart 和 preStop

postStart：容器创建之后立刻执行，用于资源部署、环境准备等。

preStop：在容器被终止前执行，用于优雅关闭应用程序、通知其他系统等。

演示 postStart 和 preStop 用法：

```
......
containers:
- image: sample:v2
  name: war
  lifecycle:
    postStart:
      exec:
        command:
        - “cp”
        - “/sample.war”
        - “/app”
    preStop:
       httpGet:
         host: monitor.com
         path: /waring
         port: 8080
         scheme: HTTP
......
```

以上示例中，定义了一个 Pod，包含一个 JAVA 的 web 应用容器，其中设置了 PostStart 和 PreStop 回调函数。即在容器创建成功后，复制/sample.war 到/app 文件夹中。而在容器终止之前，发送 HTTP 请求到 http://monitor.com:8080/waring，

即向监控系统发送警告。

优雅的删除资源对象：

当用户请求删除含有 pod 的资源对象时（如 RC、deployment 等），K8S 为了让应用程序优雅关闭（即让应用程序完成正在处理的请求后，再关闭软件），K8S 提供两种信息通知：

1）默认：K8S 通知 node 执行 docker stop 命令，docker 会先向容器中 PID 为 1 的进程发送系统信号 SIGTERM，然后等待容器中的应用程序终止执行，如果等待时间达到设定的超时时间，或者默认超时时间（30s），会继续发送 SIGKILL 的系统信号强行 kill 掉进程。

2）使用 pod 生命周期（利用 PreStop 回调函数），它执行在发送终止信号之前。

默认情况下，所有的删除操作的优雅退出时间都在 30 秒以内。kubectl delete 命令支持--grace-period=的选项，以运行用户来修改默认值。0 表示删除立即执行，并且立即从 API 中删除 pod。在节点上，被设置了立即结束的的 pod，仍然会给一个很短的优雅退出时间段，才会开始被强制杀死。如下：

```
spec:
  containers:
  - name: nginx-demo
    image: centos:nginx
    lifecycle:
      preStop:
        exec:
          # nginx -s quit gracefully terminate while SIGTERM triggers a quick exit
          command: ["/usr/local/nginx/sbin/nginx","-s","quit"]
    ports:
      - name: http
        containerPort: 80
```

7.4.2 案例演示

```
[root@master1 ~]# vim pre-start.yaml
apiVersion: v1
kind: Pod
metadata:
  name: life-demo
spec:
  containers:
  - name: lifecycle-demo-container
    image: nginx
    imagePullPolicy: IfNotPresent
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c","echo 'lifecycle hookshandler' > /usr/share/nginx/html/test.html"]
      preStop:
        exec:
          command:
          - "/bin/sh"
          - "-c"
          - "nginx -s stop"
```

总结：pod 在整个生命周期中有非常多的用户行为

1.初始化容器完成初始化

2.主容器启动后可以做启动后钩子

3.主容器结束前可以做结束前钩子

4.在主容器运行中可以做一些健康检测，如 startupprobe、livenessprobe，readnessprobe

8 Pod 容器健康探测

8.1 为什么要对容器做探测?

在 Kubernetes 中 Pod 是最小的计算单元，而一个 Pod 又由多个容器组成，相当于每个容器就是一个应用，应用在运行期间，可能因为某些意外情况致使程序挂掉。那么如何监控这些容器状态稳定性，保证服务在运行期间不会发生问题，发生问题后进行重启等机制，就成为了重中之重的事情，考虑到这点 kubernetes 推出了活性探针机制。有了存活性探针能保证程序在运行中如果挂掉能够自动重启，但是还有个经常遇到的问题，比如说，在 Kubernetes 中启动 Pod，显示明明 Pod 已经启动成功，且能访问里面的端口，但是却返回错误信息。还有就是在执行滚动更新时候，总会出现一段时间，Pod 对外提供网络访问，但是访问却发生 404，这两个原因，都是因为 Pod 已经成功启动，但是 Pod 的的容器中应用程序还在启动中导致，考虑到这点 Kubernetes 推出了就绪性探针机制。

8.1.1 默认的健康检查

Kubernetes 默认的健康检查机制:每个容器启动时都会执行一个进程,此进程由 Dockerfile 的 CMD 或 ENTRYPOINT 指定。如果进程退出时返回码非零，则认为容器发生故障，Kubernetes 就会根据 restartPolicy 策略决定是否重启容器。

```
[root@master1 ~]# vim check.yaml
apiVersion: v1
kind: Pod
metadata:
  name: check
  namespace: default
  labels:
    app: check
spec:
  containers:
  - name: check
    image: busybox:1.28
    imagePullPolicy: IfNotPresent
    command:
    - /bin/sh
    - -c
    - sleep 10;exit
[root@master1 ~]# kubectl get pods -w
NAME        READY   STATUS    RESTARTS   AGE
check       0/1     Pending   0          0s
check       0/1     Pending   0          0s
check       0/1     ContainerCreating   0          0s
check       0/1     ContainerCreating   0          1s
check       1/1     Running             0          1s
```

```
check     0/1     Completed           0          11s
check     1/1     Running             1 (1s ago)   12s
check     0/1     Completed           1 (11s ago)   22s
check     0/1     CrashLoopBackOff    1 (13s ago)   34s
check     1/1     Running             2 (13s ago)   34s
check     0/1     Completed           2 (23s ago)   44s
```

在上面的例子中，容器进程返回值非零，Kubernetes 则认为容器发生故障，需要重启。有不少情况是发生了故障，但进程并不会退出。比如访问 Web 服务器时显示 500 内部错误，可能是系统超载，也可能是资源死锁，此时 httpd 进程并没有异常退出，在这种情况下重启容器可能是最直接、最有效的解决方案。

8.1.2 k8s 提供了三种来实现容器探测的方法

1. startupProbe：探测容器中的应用是否已经启动。如果提供了启动探测(startup probe)，则禁用所有其他探测，直到它成功为止。如果启动探测失败，kubelet 将杀死容器，容器服从其重启策略进行重启。如果容器没有提供启动探测，则默认状态为成功 Success。

2. livenessprobe：用指定的方式（exec、tcp、http）检测 pod 中的容器是否正常运行，如果检测失败，则认为容器不健康，那么 Kubelet 将根据 Pod 中设置的 restartPolicy 策略来判断 Pod 是否要进行重启操作，如果容器配置中没有配置 livenessProbe，Kubelet 将认为存活探针探测一直为 success（成功）状态。

3. readinessprobe：就绪性探针，用于检测容器中的应用是否可以接受请求，当探测成功后才使 Pod 对外提供网络访问，将容器标记为就绪状态，可以加到 pod 前端负载，如果探测失败，则将容器标记为未就绪状态，会把 pod 从前端负载移除。

可以自定义在 pod 启动时是否执行这些检测，如果不设置，则检测结果均默认为通过，如果设置，则顺序为：startupProbe --> readinessProbe 和 livenessProbe，readinessProbe 和 livenessProbe 是并发关系。

目前 LivenessProbe 和 ReadinessProbe、startupprobe 探测都支持下面三种探针：

1、exec：在容器中执行指定的命令，如果执行成功，退出码为 0 则探测成功。

2、TCPSocket：通过容器的 IP 地址和端口号执行 TCP 检查，如果能够建立 TCP 连接，则表明容器健康。

3、HTTPGet：通过容器的 IP 地址、端口号及路径调用 HTTP Get 方法，如果响应的状态码大于等于 200 且小于 400，则认为容器健康。

探针探测结果有以下值：

1、Success：表示通过检测。

2、Failure：表示未通过检测。

3、Unknown：表示检测没有正常进行。

Pod 探针相关的属性：探针(Probe)有许多可选字段，可以用来更加精确的控制 Liveness 和 Readiness 两种探针的行为。

initialDelaySeconds：容器启动后要等待多少秒后探针开始工作，单位“秒”，默认是 0 秒，最小值是 0

periodSeconds：执行探测的时间间隔（单位是秒），默认为 10s，单位“秒”，最小值是 1

timeoutSeconds：探针执行检测请求后，等待响应的超时时间，默认为 1，单位“秒”

successThreshold：连续探测几次成功，才认为探测成功，默认为 1，在 Liveness 探针中必须为 1，最小值为 1

failureThreshold：探测失败的重试次数，重试一定次数后将认为失败，在 readiness 探针中，Pod 会被标记为未就绪，默认为 3，最小值为 1

两种探针区别：readinessProbe 和 livenessProbe 可以使用相同探测方式，只是对 Pod 的处置方式不同。

readinessProbe 当检测失败后，将 Pod 的 IP:Port 从对应的 EndPoint 列表中删除。

livenessProbe 当检测失败后，将杀死容器并根据 Pod 的重启策略来决定作出对应的措施。

8.2 启动探测 startupprobe

8.2.1 exec 模式

```
[root@master1 ~]# vim startup-exec.yaml
apiVersion: v1
kind: Pod
metadata:
  name: startupprobe
spec:
  containers:
  - name: startup
    image: tomcat:8.5-jre8-alpine
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 8080
    startupProbe:
      exec:
        command:
        - "/bin/sh"
        - "-c"
        - "aa ps aux | grep tomcat"    #执行会失败，所以 pod 无法 ready
      initialDelaySeconds: 20   #容器启动后多久开始探测
      periodSeconds: 20    #执行探测的时间间隔
      timeoutSeconds: 10   #探针执行检测请求后，等待响应的超时时间
      successThreshold: 1  #成功多少次才算成功
      failureThreshold: 3  #失败多少次才算失败
```

8.2.2 tcpsocket 模式

```
[root@master1 ~]# vim startup-tcpsocket.yaml
apiVersion: v1
kind: Pod
metadata:
  name: startupprobe
spec:
  containers:
  - name: startup
    image: tomcat:8.5-jre8-alpine
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 8080
    startupProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 20    #容器启动后多久开始探测
      periodSeconds: 20      #执行探测的时间间隔
```

```
      timeoutSeconds: 10     #探针执行检测请求后，等待响应的超时时间
      successThreshold: 1    #成功多少次才算成功
      failureThreshold: 3    #失败多少次才算失败
```

8.2.3 httpget 模式

```
[root@master1 ~]# vim startup-httpget.yaml
apiVersion: v1
kind: Pod
metadata:
  name: startupprobe
spec:
  containers:
  - name: startup
    image: tomcat:8.5-jre8-alpine
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 8080
    startupProbe:
      httpGet:
        path: /
        port: 8080
      initialDelaySeconds: 20     #容器启动后多久开始探测
      periodSeconds: 20      #执行探测的时间间隔
      timeoutSeconds: 10     #探针执行检测请求后，等待响应的超时时间
      successThreshold: 1    #成功多少次才算成功
      failureThreshold: 3    #失败多少次才算失败
```

8.3 存活性探测 livenessProbe

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

8.3.1 通过 exec 方式做健康探测

```
[root@master1 ~]# vim liveness-exec.yaml
apiVersion: v1
kind: Pod
metadata:
  name: liveness-exec
  labels:
    app: liveness
spec:
  containers:
  - name: liveness
    image: busybox:1.28
    imagePullPolicy: IfNotPresent
    args:                          #创建测试探针探测的文件
    - /bin/sh
```

```
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      initialDelaySeconds: 10   #延迟检测时间
      periodSeconds: 5          #检测时间间隔
      exec:
        command:
        - cat
        - /tmp/healthy
```

容器启动设置执行的命令：/bin/sh -c "touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600"

容器在初始化后，首先创建一个 /tmp/healthy 文件，然后执行睡眠命令，睡眠 30 秒，到时间后执行删除 /tmp/healthy 文件命令。而设置的存活探针检检测方式为执行 shell 命令，用 cat 命令输出 healthy 文件的内容，如果能成功执行这条命令，存活探针就认为探测成功，否则探测失败。在前 30 秒内，由于文件存在，所以存活探针探测时执行 cat /tmp/healthy 命令成功执行。30 秒后 healthy 文件被删除，所以执行命令失败，Kubernetes 会根据 Pod 设置的重启策略来判断，是否重启 Pod。

8.3.2 通过 HTTP 方式做健康探测

```
[root@node1 ~]# ctr -n k8s.io images import springboot.tar.gz
[root@node2 ~]# ctr -n k8s.io images import springboot.tar.gz
[root@master1 ~]# vim liveness-http.yaml
apiVersion: v1
kind: Pod
metadata:
  name: liveness-http
  labels:
    test: liveness
spec:
  containers:
  - name: liveness
    image: mydlqclub/springboot-helloworld:0.0.1
    imagePullPolicy: IfNotPresent
    livenessProbe:
      initialDelaySeconds: 20   #延迟加载时间
      periodSeconds: 5          #重试时间间隔
      timeoutSeconds: 10        #超时时间设置
      httpGet:
        scheme: HTTP
        port: 8081
        path: /actuator/health
```

上面 Pod 中启动的容器是一个 SpringBoot 应用，其中引用了 Actuator 组件，提供了/actuator/health 健康检查地址，存活探针可以使用 HTTPGet 方式向服务发起请求，请求 8081 端口的 /actuator/health 路径来进行存活判断：

任何大于或等于 200 且小于 400 的代码表示探测成功。

任何其他代码表示失败。如果探测失败，则会杀死 Pod 进行重启操作。

httpGet 探测方式有如下可选的控制字段：

scheme：用于连接 host 的协议，默认为 HTTP。

host：要连接的主机名，默认为 Pod IP，可以在 http request head 中设置 host 头部。

port：容器上要访问端口号或名称。

path：http 服务器上的访问 URI。

httpHeaders：自定义 HTTP 请求 headers，HTTP 允许重复 headers。

8.3.3 通过 TCP 方式做健康探测

```
[root@node1 ~]# vim liveness-tcp.yaml
apiVersion: v1
kind: Pod
metadata:
  name: liveness-tcp
  labels:
    app: liveness
spec:
  containers:
  - name: liveness
    image: nginx
    imagePullPolicy: IfNotPresent
    livenessProbe:
      initialDelaySeconds: 15
      periodSeconds: 20
      tcpSocket:
        port: 80
```

TCP 检查方式和 HTTP 检查方式非常相似，在容器启动 initialDelaySeconds 参数设定的时间后，kubelet 将发送第一个 livenessProbe 探针，尝试连接容器的 80 端口，如果连接失败则将杀死 Pod 重启容器。

8.4 就绪性探测 readinessProbe

Pod 的 ReadinessProbe 探针使用方式和 LivenessProbe 探针探测方法一样，也是支持三种，只是一个是用于探测应用的存活，一个是判断是否对外提供流量的条件。这里用一个 Springboot 项目，设置 ReadinessProbe 探测 SpringBoot 项目的 8081 端口下的/actuator/health 接口，如果探测成功则代表内部程序以及启动，就开放对外提供接口访问，否则内部应用没有成功启动，暂不对外提供访问，直到就绪探针探测成功。

8.4.1 exec 探测

```
[root@node1 ~]# vim readiness-exec.yaml
apiVersion: v1
kind: Service
metadata:
  name: springboot
  labels:
    app: springboot
spec:
  type: NodePort
  ports:
  - name: server
    port: 8080
```

```
    targetPort: 8080
    nodePort: 31180
  - name: management
    port: 8081
    targetPort: 8081
    nodePort: 31181
  selector:
    app: springboot
---
apiVersion: v1
kind: Pod
metadata:
  name: springboot
  labels:
    app: springboot
spec:
  containers:
  - name: springboot
    image: mydlqclub/springboot-helloworld:0.0.1
    imagePullPolicy: IfNotPresent
    ports:
    - name: server
      containerPort: 8080
    - name: management
      containerPort: 8081
    readinessProbe:
      initialDelaySeconds: 20
      periodSeconds: 5
      timeoutSeconds: 10
      httpGet:
        scheme: HTTP
        port: 8081
        path: /actuator/health
```

8.5 ReadinessProbe + LivenessProbe +StartupProbe 配合使用示例

一般程序中需要设置三种探针结合使用，并且也要结合实际情况，来配置初始化检查时间和检测间隔，下面列一个简单的 SpringBoot 项目的例子。

```
[root@master1 ~]# vim start-read-live.yaml
apiVersion: v1
kind: Service
metadata:
  name: springboot-live
  labels:
    app: springboot
```

```
spec:
  type: NodePort
  ports:
  - name: server
    port: 8080
    targetPort: 8080
    nodePort: 31180
  - name: management
    port: 8081
    targetPort: 8081
    nodePort: 31181
  selector:
    app: springboot
---
apiVersion: v1
kind: Pod
metadata:
  name: springboot-live
  labels:
    app: springboot
spec:
  containers:
  - name: springboot
    image: mydlqclub/springboot-helloworld:0.0.1
    imagePullPolicy: IfNotPresent
    ports:
    - name: server
      containerPort: 8080
    - name: management
      containerPort: 8081
    readinessProbe:
      initialDelaySeconds: 20
      periodSeconds: 5
      timeoutSeconds: 10
      httpGet:
        scheme: HTTP
        port: 8081
        path: /actuator/health
    livenessProbe:
      initialDelaySeconds: 20
      periodSeconds: 5
      timeoutSeconds: 10
      httpGet:
        scheme: HTTP
```

```
      port: 8081
      path: /actuator/health
  startupProbe:
    initialDelaySeconds: 20
    periodSeconds: 5
    timeoutSeconds: 10
    httpGet:
      scheme: HTTP
      port: 8081
      path: /actuator/health
```

9 K8s 控制器 Replicaset

前面我们学习了 Pod，那我们在定义 pod 资源时，可以直接创建一个 kind：Pod 类型的自主式 pod，但是这存在一个问题，假如 pod 被删除了，那这个 pod 就不能自我恢复，就会彻底被删除，线上这种情况非常危险，所以今天就给大家讲解下 pod 的控制器，所谓控制器就是能够管理 pod，监测 pod 运行状况，当 pod 发生故障，可以自动恢复 pod。也就是说能够代我们去管理 pod 中间层，并帮助我们确保每一个 pod 资源始终处于我们所定义或者我们所期望的目标状态，一旦 pod 资源出现故障，那么控制器会尝试重启 pod 或者里面的容器，如果一直重启有问题的话那么它可能会基于某种策略来进行重新布派或者重新编排；如果 pod 副本数量低于用户所定义的目标数量，它也会自动补全；如果多余，也会自动终止 pod 资源。

9.1 Replicaset 控制器解读

9.1.1 Replicaset 概述

ReplicaSet 是 kubernetes 中的一种副本控制器，简称 rs，主要作用是控制由其管理的 pod，使 pod 副本的数量始终维持在预设的个数。它的主要作用就是保证一定数量的 Pod 能够在集群中正常运行，它会持续监听这些 Pod 的运行状态，在 Pod 发生故障时重启 pod，pod 数量减少时重新运行新的 Pod 副本。官方推荐不要直接使用 ReplicaSet，用 Deployments 取而代之，Deployments 是比 ReplicaSet 更高级的概念，它会管理 ReplicaSet 并提供很多其它有用的特性，最重要的是 Deployments 支持声明式更新，声明式更新的好处是不会丢失历史变更。所以 Deployment 控制器不直接管理 Pod 对象，而是由 Deployment 管理 ReplicaSet，再由 ReplicaSet 负责管理 Pod 对象。

9.1.2 Replicaset 工作原理

Replicaset 核心作用在于代用户创建指定数量的 pod 副本，并确保 pod 副本一直处于满足用户期望的数量，起到多退少补的作用，并且还具有自动扩容缩容等机制。

Replicaset 控制器主要由三个部分组成：

1. 用户期望的 pod 副本数：用来定义由这个控制器管控的 pod 副本有几个。
2. 标签选择器：选定哪些 pod 是自己管理的。
3. pod 资源模板：如果集群中现存的 pod 数量不够我们定义的副本中期望的数量怎么办，需要新建 pod，这就需要 pod 模板，新建的 pod 是基于模板来创建的。

9.2 Replicaset 资源清单文件编写技巧

```
[root@master1 ~]# kubectl explain rs     //查看定义 Replicaset 资源需要的字段有哪些?
KIND:     ReplicaSet
VERSION:  apps/v1
FIELDS:
   apiVersion <string>  #当前资源使用的 api 版本，跟 VERSION:  apps/v1 保持一致
   kind   <string>     #资源类型，跟 KIND: ReplicaSet 保持一致
```

```
   metadata    <Object> #元数据，定义 Replicaset 名字的
   spec    <Object>     ##定义副本数、定义标签选择器、定义 Pod 模板
   status <Object> #状态信息，不能改
[root@master1 ~]# kubectl explain rs.spec     //查看 replicaset 的 spec 字段如何定义？
KIND:     ReplicaSet
VERSION:  apps/v1
RESOURCE: spec <Object>
FIELDS:
   minReadySeconds <integer>
   replicas    <integer>  #定义的 pod 副本数，根据我们指定的值创建对应数量的 pod
   selector    <Object> -required-  #用于匹配 pod 的标签选择器
   template    <Object>      #定义 Pod 的模板，基于这个模板定义的所有 pod 是一样的
[root@master1 ~]# kubectl explain rs.spec.template     //查看 replicaset 的 spec.template 字段如何定义？
KIND:     ReplicaSet
VERSION:  apps/v1
RESOURCE: template <Object>
FIELDS:
   metadata    <Object>
   spec    <Object>
[root@master1 ~]# kubectl explain rs.spec.template.spec
```

通过上面可以看到，ReplicaSet 资源中有两个 spec 字段。第一个 spec 声明的是 ReplicaSet 定义多少个 Pod 副本（默认将仅部署 1 个 Pod）、匹配 Pod 标签的选择器、创建 pod 的模板。第二个 spec 是 spec.template.spec：主要用于 Pod 里的容器属性等配置。

.spec.template 里的内容是声明 Pod 对象时要定义的各种属性，所以这部分也叫做 PodTemplate（Pod 模板）。还有一个值得注意的地方是：在.spec.selector 中定义的标签选择器必须能够匹配到 spec.template.metadata.labels 里定义的 Pod 标签，否则 Kubernetes 将不允许创建 ReplicaSet。

9.3 Replicaset 使用案例

```
[root@master1 ~]# vim replicaset.yaml      //编写一个 ReplicaSet 资源清单
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  namespace: default
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier1: frontend1
  template:
    metadata:
```

```
      labels:
        tier1: frontend1
    spec:
      containers:
      - name: php-redis
        image: nginx
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
[root@master1 ~]# kubectl apply -f replicaset.yaml
[root@master1 ~]# kubectl get rs
NAME       DESIRED   CURRENT   READY   AGE
frontend      3         3           3        53m
[root@master1 ~]# kubectl get pods
frontend-82p9b   1/1     Running   0          36m
frontend-j6twz   1/1     Running   0          36m
frontend-lcnq6   1/1     Running   0          36m
```

9.4 Replicaset 管理 pod

9.4.1 Replicaset 实现 pod 的动态扩容

ReplicaSet 最核心的功能是可以动态扩容和回缩，如果我们觉得两个副本太少了，想要增加，只需要修改配置文件 replicaset.yaml 里的 replicas 的值即可，原来 3，现在变成 4，修改之后执行如下命令更新：

```
[root@master1 ~]# kubectl apply -f replicaset.yaml
[root@master1 ~]# kubectl get rs
NAME       DESIRED   CURRENT   READY   AGE
frontend       4            4         4        62m
[root@master1 ~]# kubectl get pods
frontend-82p9b    1/1     Running   0          62m
frontend-j6twz    1/1     Running   0          62m
frontend-kzjm7    1/1     Running   0          33s
frontend-lcnq6    1/1     Running   0          62m
```

9.4.2 Replicaset 实现 pod 的动态缩容

如果我们觉得 5 个 Pod 副本太多了，想要减少，只需要修改配置文件 replicaset.yaml 里的 replicas 的值即可，把 4 变成 2，修改之后执行如下命令更新：

```
[root@master1 ~]# kubectl apply -f replicaset.yaml
[root@master1 ~]# kubectl get rs
NAME       DESIRED   CURRENT   READY   AGE
frontend   2         2         2       70m
[root@master1 ~]# kubectl get pods
frontend-j6twz   1/1     Running   0          70m
frontend-lcnq6   1/1     Running   0          70m
```

9.4.3 Replicaset 实现 pod 的更新

把 myapp-v2.tar.gz 上传到 node1 和 node2 上，手动解压

```
[root@node1 ~]# ctr -n=k8s.io images import myapp-v2.tar.gz
```

```
[root@node2 ~]# ctr -n=k8s.io images import myapp-v2.tar.gz

[root@master1 ~]# vim replicaset.yaml
修改镜像，变成 ikubernetes/myapp:v2
[root@master1 ~]# kubectl apply -f replicaset.yaml
[root@master1 ~]# kubectl get pods -o wide
NAME             READY   STATUS    RESTARTS   AGE    IP               NODE
frontend-glb2c   1/1     Running   0          34s    10.244.209.133   node1
frontend-lck9t   1/1     Running   0          34s    10.244.187.74    node2
[root@master1 ~]# curl 10.244.209.133
[root@master1 ~]# curl 10.244.209.74
```

上面可以看到，虽然 replicaset.yaml 修改了镜像，执行了 kubectl apply -f replicaset.yaml，但是 pod 还是用的 nginx 这个镜像，没有实现自动更新。

```
[root@master1 ~]# kubectl delete pods frontend-glb2c     //把 10.244.209.133 这个 ip 对应的 pod 删除
[root@master1 ~]# kubectl get pods -o wide
NAME             READY   STATUS    RESTARTS   AGE     IP               NODE
frontend-hkhdw   1/1     Running   0          15s     10.244.187.75    node2
frontend-lck9t   1/1     Running   0          2m37s   10.244.187.74    node2
```

重新生成了一个新的 pod：frontend-hkhdw

```
[root@master1 ~]# curl 10.244.187.75
Hello MyApp | Version: v2 | <a href="hostname.html">Pod Name</a>
```

新生成的 pod 的镜像已经变成了 myapp 的，说明更新完成了。

总结：生产环境如果升级，可以删除一个 pod，观察一段时间之后没问题再删除另一个 pod，但是这样需要人工干预多次；实际生产环境一般采用蓝绿发布，原来有一个 rs1，再创建一个 rs2（控制器），通过修改 service 标签，修改 service 可以匹配到 rs2 的控制器，这样才是蓝绿发布，这个也需要我们精心的部署规划，我们有一个控制器就是建立在 rs 之上完成的，叫做 Deployment。

10 Deployment 控制器

Deployment 官方文档：

https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

10.1 Deployment 概述

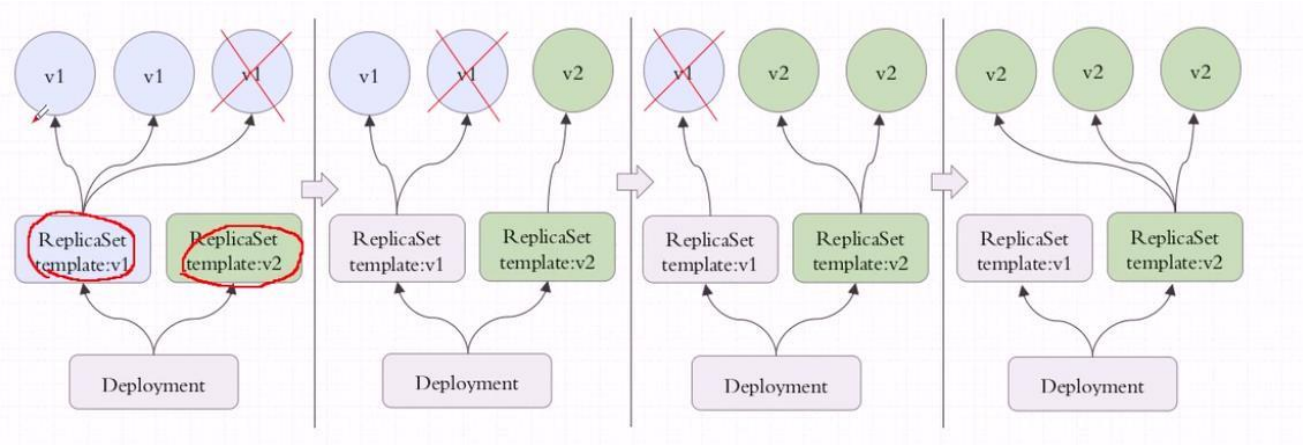
Deployment 是 kubernetes 中最常用的资源对象，为 ReplicaSet 和 Pod 的创建提供了一种声明式的定义方法，在 Deployment 对象中描述一个期望的状态，Deployment 控制器就会按照一定的控制速率把实际状态改成期望状态，通过定义一个 Deployment 控制器会创建一个新的 ReplicaSet 控制器，通过 ReplicaSet 创建 pod，删除 Deployment 控制器，也会删除 Deployment 控制器下对应的 ReplicaSet 控制器和 pod 资源。

使用 Deployment 而不直接创建 ReplicaSet 是因为 Deployment 对象拥有许多 ReplicaSet 没有的特性，例如滚动升级、金丝雀发布、蓝绿部署和回滚。

扩展：声明式定义是指直接修改资源清单文件，然后通过 kubectl apply -f 资源清单 yaml 文件，就可以更改资源。

Deployment 控制器是建立在 rs 之上的一个控制器，可以管理多个 rs，每次更新镜像版本，都会生成一个新的 rs，把旧的 rs 替换掉，多个 rs 同时存在，但是只有一个 rs 运行。

rs v1 控制三个 pod，删除一个 pod，在 rs v2 上重新建立一个，依次类推，直到全部都是由 rs v2 控制，如果 rs v2 有问题，还可以回滚，Deployment 是建构在 rs 之上的，多个 rs 组成一个 Deployment，但是只有一个 rs 处于活跃状态。



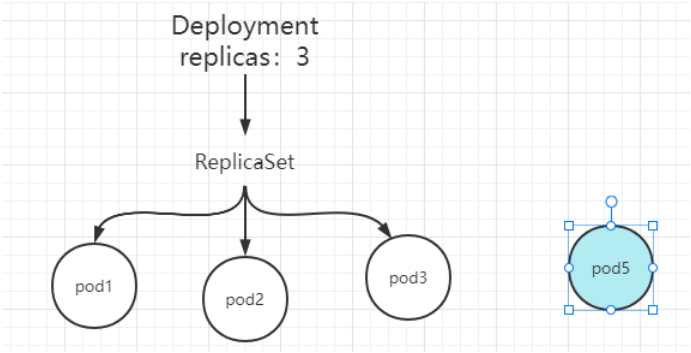
10.2 Deployment 工作原理

Deployment 可以使用声明式定义，直接在命令行通过纯命令的方式完成对应资源版本的内容的修改，也就是通过打补丁的方式进行修改；Deployment 能提供滚动式自定义自控制的更新；对 Deployment 来讲，我们在实现更新时还可以实现控制更新节奏和更新逻辑。

什么叫做更新节奏和更新逻辑呢？

比如说 Deployment 控制 5 个 pod 副本，pod 的期望值是 5 个，但是升级的时候需要额外多几个 pod，那我们控制器可以控制在 5 个 pod 副本之外还能再增加几个 pod 副本；比方说能多一个，但是不能少，那么升级的时候就是先增加一个，再删除一个，增加一个删除一个，始终保持 pod 副本数是 5 个；还有一种情况，最多允许多一个，最少允许少一个，也就是最多 6 个，最少 4 个，第一次加一个，删除两个，第二次加两个，删除两个，依次类推，可以自己控制更新方式，这种滚动更新需要加 readinessProbe 和 livenessProbe 探测，确保 pod 中容器里的应用都正常启动了才删除之前的 pod。

启动第一步，刚更新第一批就暂停了也可以；假如目标是 5 个，允许一个也不能少，允许最多可以 10 个，那一次加 5 个即可；这就是我们可以自己控制节奏来控制更新的方法。



通过 Deployment 对象，你可以轻松的做到以下事情：

1. 创建 ReplicaSet 和 Pod
2. 滚动升级（不停止旧服务的状态下升级）和回滚应用（将应用回滚到之前的版本）
3. 平滑地扩容和缩容
4. 暂停和继续 Deployment

10.3 Deployment 资源清单文件编写技巧

```
[root@master1 ~]# kubectl explain deployment     //查看 Deployment 资源对象由哪几部分组成
KIND:     Deployment
VERSION:  apps/v1
FIELDS:
```

apiVersion <string> #该资源使用的 api 版本

kind <string> #创建的资源是什么？

metadata <Object> #元数据，包括资源的名字和名称空间

spec <Object> #定义容器的

status <Object> #状态，不可以修改

[root@master1 ~]# **kubectl explain deployment.spec** //查看 Deployment 下的 spec 字段

KIND: Deployment

VERSION: apps/v1

RESOURCE: spec <Object>

FIELDS:

minReadySeconds <integer> #Kubernetes 在等待设置的时间后才进行升级，如果没有设置该值，Kubernetes 会假设该容器启动起来后就提供服务了

paused <boolean> #暂停，当我们更新的时候创建 pod 先暂停，不是立即更新

progressDeadlineSeconds <integer> #k8s 在升级过程中有可能由于各种原因升级卡住（这个时候还没有明确的升级失败），比如在拉取被墙的镜像，权限不够等错误。那么这个时候就需要有个 deadline ，在 deadline 之内如果还卡着，那么就上报这个情况，这个时候这个 Deployment 状态就被标记为 False，并且注明原因。但是它并不会阻止 Deployment 继续进行卡住后面的操作。完全由用户进行控制。

replicas <integer> #副本数

revisionHistoryLimit <integer> #保留的历史版本，默认是 10

selector <Object> -required- #标签选择器，选择它关联的 pod

strategy <Object> #更新策略

template <Object> -required #定义的 pod 模板

[root@master1 ~]# **kubectl explain deploy.spec.strategy** //查看 Deployment 下的 spec.strategy 字段

KIND: Deployment

VERSION: apps/v1

RESOURCE: strategy <Object>

FIELDS:

rollingUpdate <Object>

type <string>

Type of deployment. Can be "Recreate" or "RollingUpdate". Default is RollingUpdate.

#支持两种更新，Recreate 和 RollingUpdate，Recreate 是重建式更新，删除一个更新一个

#RollingUpdate 滚动更新，定义滚动更新方式，也就是 pod 能多几个，少几个

[root@master1 ~]# **kubectl explain deploy.spec.strategy.rollingUpdate** //查看 Deployment 下的 spec.strategy.rollingUpdate 字段

KIND: Deployment

VERSION: apps/v1

RESOURCE: rollingUpdate <Object>

FIELDS:

maxSurge <string> #我们更新的过程当中最多允许超出的指定的目标副本数有几个；它有两种取值方式，第一种直接给定数量，第二种根据百分比，百分比表示原本是 5 个，最多可以超出 20%，那就允许多一个，最多可以超过 40%，那就允许多两个。

maxUnavailable <string> #最多允许几个不可用，假设有 5 个副本，最多一个不可用，就表示最少有 4 个可用

replicas： 5

maxSurge: 25% 5*25%=1.25 ->5+2=7

```
maxUnavailable: 25%    5%25%=1.25  -> 5-1=4
[root@master1 ~]# kubectl explain deploy.spec.template    //查看 Deployment 下的 spec.template 字段

KIND:     Deployment
VERSION:  apps/v1
RESOURCE: template <Object>
FIELDS:
   metadata    <Object>  #定义模板的名字
   spec    <Object>    #deployment.spec.template 为 Pod 定义的模板，和 Pod 定义不太一样，template 中不包含 apiVersion 和 Kind 属性，要求必须有 metadata。deployment.spec.template.spec 为容器的属性信息，其他定义内容和 Pod 一致。
```

10.4 Deployment 使用案例

10.4.1 创建一个 web 站点

deployment 是一个三级结构，deployment 管理 replicaset，replicaset 管理 pod。

把 myapp-blue-v1.tar.gz 和 myapp-blue-v2.tar.gz 上传到 node1 和 node2 上，手动解压：

```
[root@node1 ~]# ctr -n=k8s.io images import myapp-blue-v1.tar.gz
[root@node2 ~]# ctr -n=k8s.io images import myapp-blue-v1.tar.gz
[root@node1 ~]# ctr -n=k8s.io images import myapp-blue-v2.tar.gz
[root@node2 ~]# ctr -n=k8s.io images import myapp-blue-v2.tar.gz
[root@master1 ~]# vim deploy-demo.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-v1
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
      version: v1
  template:
    metadata:
      labels:
        app: myapp
        version: v1
    spec:
      containers:
      - name: myapp
        image: janakiramm/myapp:v1
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
[root@master1 ~]# kubectl apply -f deploy-demo.yaml    //更新资源清单文件
```

```
[root@master1 ~]# kubectl get deploy
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
myapp-v1   2/2     2            2           60s
[root@master1 ~]# kubectl get rs
AME                     DESIRED   CURRENT   READY   AGE
myapp-v1-67fd9fc9c8     2         2            2    2m35s
[root@master1 ~]# kubectl get pods -o wide | grep myapp
myapp-v1-67fd9fc9c8-fcprr   1/1   Running   0   10.244.187.78   node2
myapp-v1-67fd9fc9c8-hw4f9   1/1   Running   0   10.244.209.136  node1
[root@master1 ~]# curl 10.244.187.78
      ...
      background-color: blue;
[root@master1 ~]# curl 10.244.209.136
      ...
      background-color: blue;
```

10.5 Deployment 管理 pod

10.5.1 通过 deployment 管理应用，实现扩容，把副本数变成 3

```
[root@master1 ~]# vim deploy-demo.yaml
```

直接修改 replicas 数量，变成 3

```
spec:
  replicas: 3
```

修改之后保存退出，执行：

```
[root@master1 ~]# kubectl apply -f deploy-demo.yaml
```

注意：apply 不同于 create，apply 可以执行多次；create 执行一次，再执行就会报错复。

```
[root@master1 ~]# kubectl get pods
myapp-v1-67fd9fc9c8-fcprr   1/1     Running   0          15m
myapp-v1-67fd9fc9c8-h9js5   1/1     Running   0          11s
myapp-v1-67fd9fc9c8-hw4f9   1/1     Running   0          17m
[root@master1 ~]# kubectl describe deploy myapp-v1
```

10.5.2 通过 deployment 管理应用，实现缩容，把副本数变成 2

```
[root@master1 ~]# vim deploy-demo.yaml
```

直接修改 replicas 数量，如下，变成 2

```
spec:
  replicas: 2
```

修改之后保存退出，执行：

```
[root@master1 ~]# kubectl apply -f deploy-demo.yaml
[root@master1 ~]# kubectl get pods
myapp-v1-67fd9fc9c8-fcprr    1/1     Running   0          18m
myapp-v1-67fd9fc9c8-hw4f9    1/1     Running   0          20m
```

10.6 通过 k8s 实现滚动更新

10.6.1 滚动更新简介

滚动更新是一种自动化程度较高的发布方式，用户体验比较平滑，是目前成熟型技术组织所采用的主流发布方式，一次

滚动发布一般由若干个发布批次组成，每批的数量一般是可以配置的（可以通过发布模板定义），例如第一批 1 台，第二批 10%，第三批 50%，第四批 100%。每个批次之间留观察间隔，通过手工验证或监控反馈确保没有问题再发下一批次，所以总体上滚动式发布过程是比较缓慢的。

10.6.2 在 k8s 中实现滚动更新

看下 Deployment 资源对象的组成：

```
[root@master1 ~]# kubectl explain deployment
[root@master1 ~]# kubectl explain deployment.spec
 KIND:     Deployment
 VERSION:  apps/v1
 RESOURCE: spec <Object>
 FIELDS:
    minReadySeconds <integer>
    paused <boolean>  #暂停，当我们更新的时候创建 pod 先暂停，不是立即更新。
    progressDeadlineSeconds  <integer>
    replicas    <integer>
    revisionHistoryLimit <integer>   #保留的历史版本数，默认是 10 个
    selector    <Object> -required-
    strategy    <Object>  #更新策略，支持的滚动更新策略
    template    <Object> -required-
[root@master1 ~]# kubectl explain deploy.spec.strategy
 KIND:     Deployment
 VERSION:  apps/v1
 RESOURCE: strategy <Object>
 FIELDS:
    rollingUpdate    <Object>
    type    <string>
 #支持两种更新，Recreate 和 RollingUpdate
 #Recreate 是重建式更新，删除一个更新一个
 #RollingUpdate 滚动更新，定义滚动更新的更新方式的，也就是 pod 能多几个，少几个，控制更新力度的。
[root@master1 ~]# kubectl explain deploy.spec.strategy.rollingUpdate
 KIND:     Deployment
 VERSION:  apps/v1
 RESOURCE: rollingUpdate <Object>
 FIELDS:
    maxSurge    <string>  #我们更新的过程当中最多允许超出的指定的目标副本数有几个；它有两种取值方式，第一种直接给定数量，第二种根据百分比，百分比表示原本是 5 个，最多可以超出 20%，那就允许多一个，最多可以超过 40%，那就允许多两个。
    maxUnavailable  <string>  #最多允许几个不可用，假设有 5 个副本，最多一个不可用，就表示最少有 4 个可用。
```

10.6.3 测试滚动更新

在终端执行如下：

```
[root@master1 ~]# kubectl get pods -l app=myapp -w
```

打开一个新的终端窗口更改镜像版本，按如下操作：

```
[root@master1 ~]# vim deploy-demo.yaml
```

把 image: janakiramm/myapp:v1 变成 image: janakiramm/myapp:v2

保存退出，执行

```
[root@master1 ~]# kubectl apply -f deploy-demo.yaml
```

再回到第一个监测的窗口，可以看到信息如下：

```
NAME                        READY   STATUS    RESTARTS   AGE
myapp-v1-67fd9fc9c8-tsl92   1/1     Running   0          22m
myapp-v1-67fd9fc9c8-4bv5n   1/1     Running   0          22m
myapp-v1-67fd9fc9c8-cw59c   1/1     Running   0          14m
myapp-v1-75fb478d6c-24tbp   0/1     Pending   0          0s
myapp-v1-75fb478d6c-24tbp   0/1     Pending   0          0s
myapp-v1-75fb478d6c-24tbp   0/1     ContainerCreating   0          0s
myapp-v1-75fb478d6c-24tbp   1/1     Running             0          11s
myapp-v1-67fd9fc9c8-cw59c   1/1     Terminating         0          15m
myapp-v1-75fb478d6c-f52l6   0/1     Pending             0          0s
myapp-v1-75fb478d6c-f52l6   0/1     Pending             0          0s
myapp-v1-75fb478d6c-f52l6   0/1     ContainerCreating   0          0s
myapp-v1-67fd9fc9c8-cw59c   0/1     Terminating         0          15m
myapp-v1-75fb478d6c-f52l6   1/1     Running             0          11s
myapp-v1-67fd9fc9c8-4bv5n   1/1     Terminating         0          23m
myapp-v1-75fb478d6c-jlw28   0/1     Pending             0          0s
myapp-v1-75fb478d6c-jlw28   0/1     Pending             0          0s
myapp-v1-75fb478d6c-jlw28   0/1     ContainerCreating   0          0s
myapp-v1-75fb478d6c-jlw28   1/1     Running             0          1s
```

pending 表示正在进行调度，ContainerCreating 表示正在创建一个 pod，running 表示运行一个 pod，running 起来一个 pod 之后再 Terminating（停掉）一个 pod，以此类推，直到所有 pod 完成滚动升级。

```
[root@master1 ~]# kubectl get rs
```

显示如下：

```
NAME                  DESIRED   CURRENT   READY   AGE
myapp-v1-75fb478d6c   3         3         3       2m7s
myapp-v1-67fd9fc9c8   0         0         0       25m
```

上面可以看到 rs 有两个，下面那个是升级之前的，已经被停掉，但是可以随时回滚。

```
[root@master1 ~]# kubectl rollout history deployment myapp-v1  //查看 myapp-v1 这个控制器的滚动历史
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
[root@master1 ~]# kubectl rollout undo deployment/myapp-v1 --to-revision=1   //回滚操作
```

10.6.4 自定义滚动更新策略

maxSurge 和 maxUnavailable 用来控制滚动更新的更新策略。

数值：

1. maxUnavailable: [0, 副本数]
2. maxSurge: [0, 副本数]

注意：两者不能同时为 0。

比例：

1. maxUnavailable: [0%, 100%] 向下取整，比如 10 个副本，5%的话==0.5 个，但计算按照 0 个；
2. maxSurge: [0%, 100%] 向上取整，比如 10 个副本，5%的话==0.5 个，但计算按照 1 个；

注意：两者不能同时为 0。

建议配置：

1. maxUnavailable == 0
2. maxSurge == 1

这是我们生产环境提供给用户的默认配置。即“一上一下，先上后下”最平滑原则：

1 个新版本 pod ready 后，才销毁旧版本 pod。此配置适用场景是平滑更新、保证服务平稳，缺点就是“太慢”了。

总结：

maxUnavailable：和期望的副本数比，不可用副本数最大比例（或最大值），这个值越小，服务越稳定，更新越平滑。

maxSurge：和期望的副本数比，超过期望副本数最大比例（或最大值），这个值调的越大，副本更新速度越快。

自定义策略：

修改更新策略：maxUnavailable=1，maxSurge=1

```
[root@master1 ~]# kubectl patch deployment myapp-v1 -p '{"spec":{"strategy":{"rollingUpdate":{"maxSurge":1,"maxUnavailable":1}}}}'
[root@master1 ~]# kubectl describe deployment myapp-v1   //查看 myapp-v1 这个控制器的详细信息
RollingUpdateStrategy:  1 max unavailable, 1 max surge
```

上面可以看到 RollingUpdateStrategy: 1 max unavailable, 1 max surge

这个 rollingUpdate 更新策略变成了刚才设定的，因为我们设定的 pod 副本数是 3，1 和 1 表示最少不能少于 2 个 pod，最多不能超过 4 个 pod ，这个就是通过控制 RollingUpdateStrategy 这个字段来设置滚动更新策略的。

把 pod 更新策略变成 Recreate:

```
[root@master1 ~]# vim deploy-demo.yaml
......
spec:
  strategy:
    type: Recreate
......
[root@master1 ~]# kubectl apply -f deploy-demo.yaml
```

打开新的终端，看 pod 更新过程:

```
[root@master1 ~]# kubectl get pods -w
```

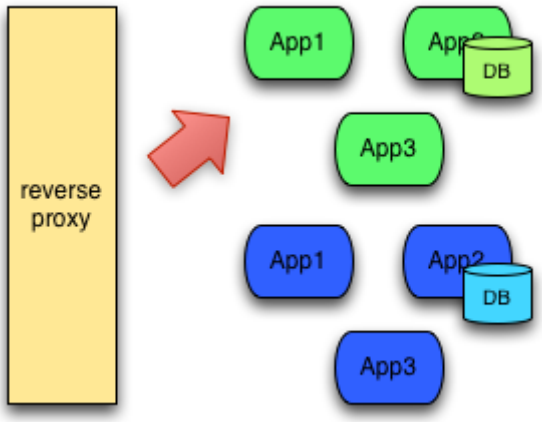
总结：recreate 这种更新策略，会把之前的所有 pod 都删除，再创建新的 pod，风险很大。

10.7 生产环境如何实现蓝绿部署？

10.7.1 什么是蓝绿部署？

蓝绿部署中，一共有两套系统：一套是正在提供服务系统，标记为“绿色”；另一套是准备发布的系统，标记为“蓝色”。两套系统都是功能完善的、正在运行的系统，只是系统版本和对外服务情况不同。

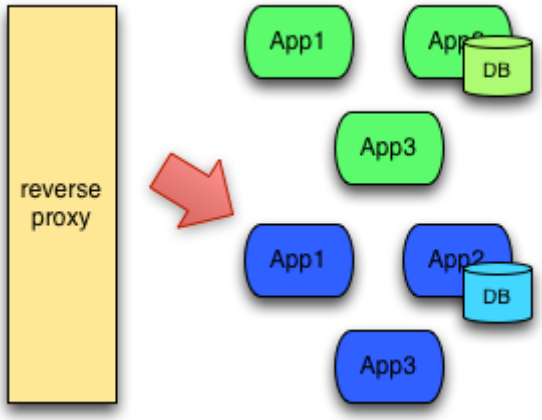
开发新版本，要用新版本替换线上的旧版本，在线上的系统之外，搭建了一个使用新版本代码的全新系统。 这时候，一共有两套系统在运行，正在对外提供服务的老系统是绿色系统，新部署的系统是蓝色系统。



蓝色系统不对外提供服务，用来做什么呢？

用来做发布前测试，测试过程中发现任何问题，可以直接在蓝色系统上修改，不干扰用户正在使用的系统。（注意，两套系统没有耦合的时候才能百分百保证不干扰）

蓝色系统经过反复的测试、修改、验证，确定达到上线标准之后，直接将用户切换到蓝色系统：



切换后的一段时间内，依旧是蓝绿两套系统并存，但是用户访问的已经是蓝色系统。这段时间内观察蓝色系统（新系统）工作状态，如果出现问题，直接切换回绿色系统。

当确信对外提供服务的蓝色系统工作正常，不对外提供服务的绿色系统已经不再需要的时候，蓝色系统正式成为对外提供服务系统，成为新的绿色系统。 原先的绿色系统可以销毁，将资源释放出来，用于部署下一个蓝色系统。

10.7.2 蓝绿部署的优势和缺点

优点：

1、更新过程无需停机，风险较少。

2、回滚方便，只需要更改路由或者切换 DNS 服务器，效率较高。

缺点：

1、成本较高，需要部署两套环境。如果新版本有问题会影响全网用户。

2、需要部署两套机器，费用开销大。

3、在非隔离的机器（Docker、VM）上操作时，可能会导致蓝绿环境被摧毁风险。

4、负载均衡器/反向代理/路由/DNS 处理不当，将导致流量没有切换过来情况出现。

10.7.3 通过 k8s 实现线上业务的蓝绿部署

把 myapp-v1.tar.gz 和 myapp-v2.tar.gz 上传到 node1 和 node2 上，手动解压：ctr -n=k8s.io images import

Kubernetes 不支持内置的蓝绿部署。目前最好的方式是创建新的 deployment，然后更新应用程序的 service 以指向新的

创建蓝色部署环境（基于第一版代码做的镜像运行的 pod）

```
[root@master1 ~]# vim blue.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-v1
  namespace: blue-green
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
      version: v1
  template:
    metadata:
      labels:
        app: myapp
        version: v1
    spec:
      containers:
      - name: myapp
        image: janakiramm/myapp:v1
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
[root@master1 ~]# kubectl create ns blue-green
[root@master1 ~]# kubectl apply -f blue.yaml
[root@master1 ~]# kubectl get pods -n blue-green
myapp-v1-75d7db5cf7-4qnjz   1/1     Running   0          8s
myapp-v1-75d7db5cf7-5sk6j   1/1     Running   0          8s
myapp-v1-75d7db5cf7-wmhs4   1/1     Running   0          8s
[root@master1 ~]# kubectl get pods -n blue-green --show-labels
NAME                     READY   STATUS   RESTARTS   AGE   LABELS
myapp-v1-75d7db5cf7-4qnjz  1/1   Running 0   35s    app=myapp,pod-template-hash=75d7db5cf7,version=v1
myapp-v1-75d7db5cf7-5sk6j  1/1   Running 0   35s    app=myapp,pod-template-hash=75d7db5cf7,version=v1
myapp-v1-75d7db5cf7-wmhs4  1/1   Running 0   5s     app=myapp,pod-template-hash=75d7db5cf7,version=v1
[root@master1 ~]# vim service_lanlv.yaml   //修改前端 service
apiVersion: v1
kind: Service
metadata:
  name: myapp-lan-lv
  namespace: blue-green
```

```
  labels:
    app: myapp
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 30062
    name: http
  selector:
    app: myapp
    version: v1
[root@master1 ~]# kubectl apply -f service_lanlv.yaml
[root@master1 ~]# kubectl get svc -n blue-green
NAME           TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
myapp-lan-lv   NodePort   10.107.213.1    <none>        80:30062/TCP   36s
```

在浏览器访问 http://k8s-master 节点 ip:30062，例如：http://192.168.40.180:30062，显示如下：

创建绿色部署环境（新上线的环境，要替代蓝色环境）

```
[root@master1 ~]# vim green.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-v2
  namespace: blue-green
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
      version: v2
  template:
    metadata:
      labels:
        app: myapp
        version: v2
    spec:
      containers:
      - name: myapp
        image: janakiramm/myapp:v2
```

```
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
```

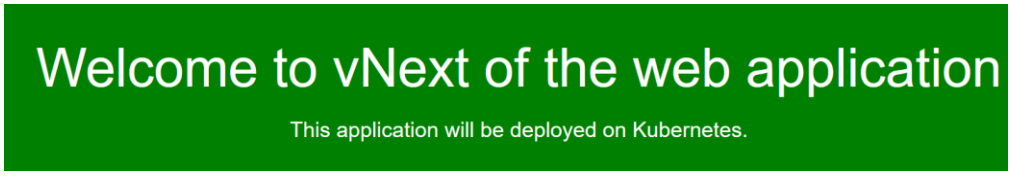
然后可以使用 kubectl 命令创建部署。

```
[root@master1 ~]# kubectl apply -f lan.yaml
[root@master1 ~]# kubectl get pods -n blue-green
NAME                          READY   STATUS    RESTARTS   AGE
myapp-v1-75d7db5cf7-4qnjz     1/1     Running   0          13m
myapp-v1-75d7db5cf7-5sk6j     1/1     Running   0          13m
myapp-v1-75d7db5cf7-wmhs4     1/1     Running   0          13m
myapp-v2-85cc897d89-hp5j2     1/1     Running   0          10s
myapp-v2-85cc897d89-jpgbm     1/1     Running   0          10s
myapp-v2-85cc897d89-q94g4     1/1     Running   0          10s
```

修改 service_lanlv.yaml 配置文件，修改标签，让其匹配到绿程序（升级之后的程序）

```
[root@master1 ~]# vim service_lanlv.yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-lan
  namespace: blue-green
  labels:
    app: myapp
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 30062
    name: http
  selector:
    app: myapp
    version: v2
[root@master1 ~]# kubectl apply -f service_lanlv.yaml  //更新资源清单文件
[root@master1 ~]# kubectl get svc -n blue-green
NAME           TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
myapp-lan-lv   NodePort   10.107.213.1    <none>        80:30062/TCP   9m50s
```

在浏览器访问 http://k8s-master 节点 ip:30062 显示如下：



实验完成之后，把资源先删除，以免影响后面实验：

```
[root@master1 ~]# kubectl delete -f blue.yaml
[root@master1 ~]# kubectl delete -f green.yaml
[root@master1 ~]# kubectl delete -f service_lanlv.yaml
```

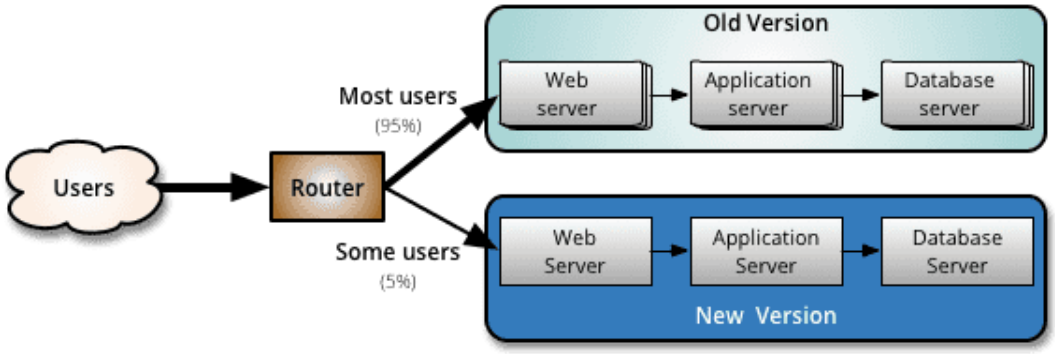
10.8 金丝雀发布

10.8.1 金丝雀发布简介

金丝雀发布的由来：17 世纪，英国矿井工人发现，金丝雀对瓦斯这种气体十分敏感。空气中哪怕有极其微量的瓦斯，金丝雀也会停止歌唱；当瓦斯含量超过一定限度时，虽然人类毫无察觉，金丝雀却早已毒发身亡。当时在采矿设备相对简陋的条件下，工人们每次下井都会带上一只金丝雀作为瓦斯检测指标，以便在危险状况下紧急撤离。

金丝雀发布（又称灰度发布、灰度更新）：金丝雀发布一般先发 1 台，或者一个小比例，例如 2%的服务器，主要做流量验证用，也称为金丝雀 (Canary) 测试 （国内常称灰度测试）。

简单的金丝雀测试一般通过手工测试验证，复杂的金丝雀测试需要比较完善的监控基础设施配合，通过监控指标反馈，观察金丝雀的健康状况，作为后续发布或回退的依据。 如果金丝测试通过，则把剩余的 V1 版本全部升级为 V2 版本。如果金丝雀测试失败，则直接回退金丝雀，发布失败。



优点：灵活，策略自定义，可以按照流量或具体的内容进行灰度，出现问题不会影响全网用户。

缺点：没有覆盖到所有的用户导致出现问题不好排查。

10.8.2 在 k8s 中实现金丝雀发布

打开一个标签 1 监测更新过程

```
[root@master1 ~]# kubectl apply -f blue.yaml
[root@master1 ~]# kubectl get pods -l app=myapp -n blue-green -w
NAME                        READY   STATUS    RESTARTS   AGE
myapp-v1-75d7db5cf7-c7z5d   1/1     Running   0          5s
myapp-v1-75d7db5cf7-km5bg   1/1     Running   0          5s
myapp-v1-75d7db5cf7-p7c8j   1/1     Running   0          5s
```

打开另一个标签 2 执行如下操作：

```
[root@master1 ~]# kubectl set image deployment myapp-v1 myapp=janakiramm/myapp:v2 -n blue-green && kubectl rollout pause deployment myapp-v1 -n blue-green
```

回到标签 1 观察，显示如下：

```
NAME                        READY   STATUS              RESTARTS   AGE
myapp-v1-67fd9fc9c8-5fd2f   1/1     Running             0          86s
myapp-v1-67fd9fc9c8-92mdr   1/1     Running             0          86s
myapp-v1-75fb478d6c-wddds   0/1     Pending             0          0s
myapp-v1-75fb478d6c-wddds   0/1     Pending             0          0s
myapp-v1-75fb478d6c-wddds   0/1     ContainerCreating   0          0s
myapp-v1-75fb478d6c-wddds   0/1     ContainerCreating   0          1s
myapp-v1-75fb478d6c-wddds   1/1     Running             0          2s
```

注：上面的解释说明把 myapp 这个容器的镜像更新到 janakiramm/myapp:v2 版本 更新镜像之后，创建一个新的 pod 就立即暂停，这就是我们说的金丝雀发布；如果暂停几个小时之后没有问题，那么取消暂停，就会依次执行后面步骤，把所有 pod 都升级。

解除暂停，回到标签 1 继续观察：

打开标签 2 执行如下：

```
[root@master1 ~]# kubectl rollout resume deployment myapp-v1 -n blue-green
```

在标签 1 可以看到如下一些信息，下面过程是把余下的 pod 里的容器都更的版本：

```
NAME                         READY   STATUS    RESTARTS   AGE
myapp-v1-67fd9fc9c8-5fd2f    1/1     Running   0          86s
myapp-v1-67fd9fc9c8-92mdr    1/1     Running   0          86s
......
myapp-v1-67fd9fc9c8-5fd2f    0/1     Terminating         0          10m
myapp-v1-67fd9fc9c8-92mdr    0/1     Terminating         0          10m
myapp-v1-67fd9fc9c8-92mdr    0/1     Terminating         0          10m
[root@master1 ~]# kubectl get rs -n blue-green
```

可以看到 replicaset 控制器有 2 个了

```
NAME                   DESIRED   CURRENT   READY   AGE
myapp-v1-67fd9fc9c8    0         0         0       13m
myapp-v1-75fb478d6c    2         2         2       7m28s
```

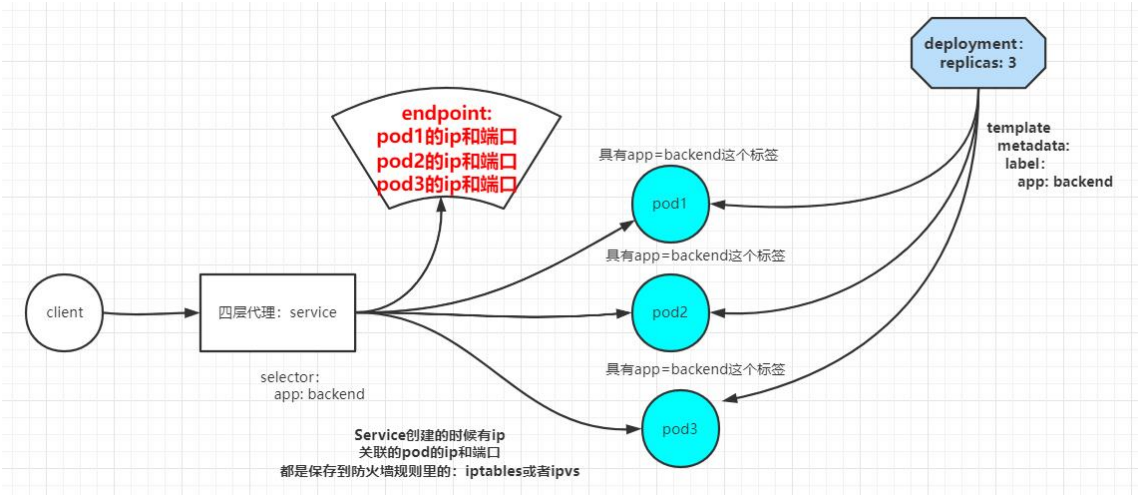
11 K8S 四层代理 Service

11.1 四层负载均衡 Service

11.1.1 为什么要有 Service?

在 kubernetes 中，Pod 是有生命周期的，如果 Pod 重启它的 IP 很有可能会发生变化。如果我们的服务都是将 Pod 的 IP 地址写死，Pod 挂掉或者重启，和刚才重启的 pod 相关联的其他服务将会找不到它所关联的 Pod，为了解决这个问题，在 kubernetes 中定义了 service 资源对象，Service 定义了一个服务访问的入口，客户端通过这个入口即可访问服务背后的应用集群实例，service 是一组 Pod 的逻辑集合，这一组 Pod 能够被 Service 访问到，通常是通过 Label Selector 实现的。

可以看下面的图：



1、pod ip 经常变化，service 是 pod 的代理，我们客户端访问，只需要访问 service，就会把请求代理到 Pod。

2、pod ip 在 k8s 集群之外无法访问，所以需要创建 service，这个 service 可以在 k8s 集群外访问的。

11.1.2 Service 概述

service 是一个固定接入层，客户端可以通过访问 service 的 ip 和端口访问到 service 关联的后端 pod，这个 service 工作依赖于在 kubernetes 集群之上部署的一个附件，就是 kubernetes 的 dns 服务（不同 kubernetes 版本的 dns 默认使用的也是不一样的，1.11 之前的版本使用的是 kubeDNs，较新的版本使用的是 coredns），service 的名称解析是依赖于 dns 附件的，因此在部署完 k8s 之后需要再部署 dns 附件，kubernetes 要想给客户端提供网络功能，需要依赖第三方的网络插件（flannel，calico 等）。每个 K8s 节点上都有一个组件叫做 kube-proxy，kube-proxy 这个组件将始终监视着 apiserver 中有关 service 资源的变动信息，需要跟 master 之上的 apiserver 交互，随时连接到 apiserver 上获取任何一个与 service 资源相关的资源变动状态，这种是通过 kubernetes 中固有的一种请求方法 watch（监视）来实现的，一旦有 service 资源的内容发生变动（如创建，删除），kube-proxy 都会将它转化成当前节点之上的能够实现 service 资源调度，把我们请求调度到后端特定的 pod 资源之上的规则，这个规则可能是 iptables，也可能是 ipvs，取决于 service 的实现方式。

11.1.3 Service 工作原理

k8s 在创建 Service 时，会根据标签选择器 selector(lable selector)来查找 Pod，据此创建与 Service 同名的 endpoint 对象，当 Pod 地址发生变化时，endpoint 也会随之发生变化，service 接收前端 client 请求的时候，就会通过 endpoint，找到转发到哪个 Pod 进行访问的地址。(至于转发到哪个节点的 Pod，由负载均衡 kube-proxy 决定)

11.1.4 kubernetes 集群中有三类 IP 地址

1、Node Network（节点网络），物理节点或者虚拟节点的网络，如 ens33 接口上的网路地址

2、Pod network（pod 网络），创建的 Pod 具有的 IP 地址

```
[root@master1 ~]# kubectl get pods -o wide
NAME             READY   STATUS          IP               NODE
frontend-h78gw   1/1     Running         10.244.187.76    node2
```

Node Network 和 Pod network 这两种网络地址是我们实实在在配置的，其中节点网络地址是配置在节点接口之上，而 pod 网络地址是配置在 pod 资源之上的，因此这些地址都是配置在某些设备之上的，这些设备可能是硬件，也可能是软件模拟的。

3、Cluster Network（集群地址，也称为 service network），这个地址是虚拟的地址（virtual ip），没有配置在某个接口上，只是出现在 service 的规则当中。

```
[root@master1 ~]# kubectl get svc
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
kubernetes        ClusterIP   10.96.0.1       <none>        443/TCP
```

11.2 Service 资源字段解释

```
[root@master1 ~]# kubectl explain service     //查看定义 Service 资源需要的字段有哪些？
KIND:     Service
VERSION:  v1
FIELDS:
   apiVersion <string>       #service 资源使用的 api 组
   kind <string>             #创建的资源类型
   metadata    <Object>      #定义元数据
   spec    <Object>
[root@master1 ~]# kubectl explain service.spec     //查看 service 的 spec 字段如何定义？
KIND:     Service
VERSION:  v1
RESOURCE: spec <Object>
FIELDS:
   allocateLoadBalancerNodePorts <boolean>
```

clusterIP <string> #动态分配的地址，也可以自己在创建的时候指定，创建之后就改不了了

clusterIPs <[]string>

externalIPs <[]string>

externalName <string>

externalTrafficPolicy <string>

healthCheckNodePort <integer>

ipFamilies <[]string>

ipFamilyPolicy <string>

loadBalancerIP <string>

loadBalancerSourceRanges <[]string>

ports <[]Object> #定义 service 端口，用来和后端 pod 建立联系

publishNotReadyAddresses <boolean>

selector <map[string]string> #通过标签选择器选择关联的 pod 有哪些

sessionAffinity <string>

sessionAffinityConfig <Object> #service 在实现负载均衡的时候还支持 sessionAffinity，sessionAffinity，什么意思？会话联系，默认是 none，随机调度的（基于 iptables 规则调度的）；如果我们定义 sessionAffinity 的 client ip，那就表示把来自同一客户端的 IP 请求调度到同一个 pod 上

topologyKeys <[]string>

type <string> #定义 service 的类型

Service 的四种类型:

[root@master1 ~]# **kubectl explain service.spec.type** //查看定义 Service.spec.type 需要的字段有哪些？

KIND: Service

VERSION: v1

FIELD: type <string>

DESCRIPTION:

type determines how the Service is exposed. Defaults to ClusterIP. Valid

options are ExternalName, ClusterIP, NodePort, and LoadBalancer.

https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types

[root@master1 ~]# **kubectl explain service.spec.ports** //查看 service 的 spec.ports 字段如何定义？

KIND: Service

VERSION: v1

RESOURCE: ports <[]Object>

FIELDS:

appProtocol <string>

name <string> #定义端口的名字

nodePort <integer> #service 在物理机映射的端口，默认在 30000-32767 之间

port <integer> -required- #service 的端口，这个是 k8s 集群内部服务可访问的端口

protocol <string>

targetPort <string> #targetPort 是 pod 上的端口，从 port 和 nodePort 上来的流量，经过 kube-proxy 流入到后端 pod 的 targetPort 上，最后进入容器。

11.3 创建 Service：type 类型是 ClusterIP

[root@master1 ~]# vim pod_test.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80  #pod 中的容器需要暴露的端口
[root@master1 ~]# kubectl apply -f pod_test.yaml
[root@master1 ~]# kubectl get pods -l run=my-nginx -o wide
NAME                            STATUS           IP                  NODE
my-nginx-5b56ccd65f-26vcz       Running          10.244.187.101      node2
my-nginx-5b56ccd65f-95n7p       Running          10.244.209.149      node1
请求 pod ip 地址，查看结果:
[root@master1 ~]# curl 10.244.187.101
<!DOCTYPE html>
<html>
<h1>Welcome to nginx!</h1>
</body>
</html>
[root@master1 ~]# curl 10.244.209.149
<!DOCTYPE html>
<html>
<h1>Welcome to nginx!</h1>
</body>
</html>
[root@master1 ~]# kubectl exec -it my-nginx-5b56ccd65f-26vcz -- /bin/bash
root@my-nginx-5b56ccd65f-26vcz:/# curl 10.244.209.149
<!DOCTYPE html>
<html>
<h1>Welcome to nginx!</h1>
</html>
```

```
root@my-nginx-5b56ccd65f-26vcz:/# exit
```

需要注意的是，pod 虽然定义了容器端口，但是不会使用调度到该节点上的 80 端口，也不会使用任何特定的 NAT 规则去路由流量到 Pod 上。这意味着可以在同一个节点上运行多个 Pod，使用相同的容器端口，并且可以从集群中任何其他的 Pod 或节点上使用 IP 的方式访问到它们。

误删除其中一个 Pod：

```
[root@master1 ~]# kubectl delete pods my-nginx-5b56ccd65f-26vcz
[root@master1 ~]# kubectl get pods -l run=my-nginx -o wide
NAME                          STATUS       IP               NODE
my-nginx-5b56ccd65f-7xzr4     Running      10.244.187.102   node2
my-nginx-5b56ccd65f-95n7p     Running      10.244.209.149   node1
```

通过上面可以看到重新生成了一个 pod：my-nginx-5b56ccd65f-7xzr4，ip 是 10.244.187.102，在 k8s 中创建 pod，如果 pod 被删除了，重新生成的 pod ip 地址会发生变化，所以需要在 pod 前端加一个固定接入层。

```
[root@master1 ~]# kubectl get pods --show-labels     //查看 pod 标签
[root@master1 ~]# vim service_test.yaml    //创建 Service
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  type: ClusterIP
  ports:
  - port: 80            #service 的端口，暴露给 k8s 集群内部服务访问
    protocol: TCP
    targetPort: 80      #pod 容器中定义的端口
  selector:
    run: my-nginx       #选择拥有 run=my-nginx 标签的 pod
```

上述 yaml 文件将创建一个 Service，具有标签 run=my-nginx 的 Pod，目标 TCP 端口 80，并且在一个抽象的 Service 端口（targetPort：容器接收流量的端口；port：抽象的 Service 端口，可以使任何其它 Pod 访问该 Service 的端口）上暴露。

```
[root@master1 ~]# kubectl apply -f service_test.yaml
[root@master1 ~]# kubectl get svc -l run=my-nginx
NAME       TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
my-nginx   ClusterIP   10.99.198.177   <none>        80/TCP     143m
```

在 k8s 控制节点访问 service 的 ip:端口就可以把请求代理到后端 pod

```
[root@master1 ~]# curl 10.99.198.177:80
<!DOCTYPE html>
<html>
<h1>Welcome to nginx!</h1>
</html>
```

通过上面可以看到请求 service IP:port 跟直接访问 pod ip:port 看到的结果一样，这就说明 service 可以把请求代理

到它所关联的后端 pod。

注意：上面的 10.99.198.177:80 地址只能是在 k8s 集群内部可以访问，在外部无法访问，比方说我们想要通过浏览器访问，那么是访问不通的，如果想要在 k8s 集群之外访问，是需要把 service type 类型改成 nodePort 的。

```
[root@master1 ~]# kubectl describe svc my-nginx     //查看 service 详细信息
Name:              my-nginx
Namespace:         default
Labels:            run=my-nginx
Annotations:       <none>
Selector:          run=my-nginx
Type:              ClusterIP
IP Families:       <none>
IP:                10.99.198.177
IPs:               10.99.198.177
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         10.244.187.102:80,10.244.209.149:80
Session Affinity:  None
Events:            <none>
[root@master1 ~]# kubectl get ep my-nginx
NAME       ENDPOINTS                                AGE
my-nginx   10.244.187.102:80,10.244.209.149:80   142m
```

service 可以对外提供统一固定的 ip 地址，并将请求重定向至集群中的 pod。其中“将请求重定向至集群中的 pod”就是通过 endpoint 与 selector 协同工作实现。selector 是用于选择 pod，由 selector 选择出来的 pod 的 ip 地址和端口号，将会被记录在 endpoint 中。endpoint 便记录了所有 pod 的 ip 地址和端口号。当一个请求访问到 service 的 ip 地址时，就会从 endpoint 中选择出一个 ip 地址和端口号，然后将请求重定向至 pod 中。具体把请求代理到哪个 pod，需要的就是 kube-proxy 的轮询实现的。service 不会直接到 pod，service 是直接到 endpoint 资源，就是地址加端口，再由 endpoint 再关联到 pod。

service 只要创建完成，我们就可以直接解析它的服务名，每一个服务创建完成后都会在集群 dns 中动态添加一个资源记录，添加完成后我们就可以解析了，资源记录格式是：

```
SVC_NAME.NS_NAME.DOMAIN.LTD.
```

服务名.命名空间.域名后缀

集群默认的域名后缀是：svc.cluster.local

就像我们上面创建的 my-nginx 这个服务，它的完整名称解析就是：my-nginx.default.svc.cluster.local

```
[root@master1 ~]# kubectl exec -it my-nginx-5b56ccd65f-7xzr4 -- /bin/bash
root@my-nginx-5b56ccd65f-7xzr4:/# curl my-nginx.default.svc.cluster.local
<!DOCTYPE html>
<h1>Welcome to nginx!</h1>
root@my-nginx-5b56ccd65f-7xzr4:/# exit
```

11.4 创建 Service：type 类型是 NodePort

```
[root@master1 ~]# vim pod_nodeport.yaml    //创建一个 pod 资源
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: my-nginx-nodeport
spec:
  selector:
    matchLabels:
      run: my-nginx-nodeport
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx-nodeport
    spec:
      containers:
      - name: my-nginx-nodeport-container
        image: nginx
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
[root@master1 ~]# kubectl apply -f pod_nodeport.yaml
[root@master1 ~]# kubectl get pods -l run=my-nginx-nodeport -owide
my-nginx-nodeport-7776f84758-9w5mh   1/1     Running   0          6m53s   10.244.121.36   node1
my-nginx-nodeport-7776f84758-h6hk8   1/1     Running   0          6m53s   10.244.102.86   node2
[root@master1 ~]# vim service_nodeport.yaml   //创建 service，代理 pod
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-nodeport
  labels:
    run: my-nginx-nodeport
spec:
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
    nodePort: 30380
  selector:
    run: my-nginx-nodeport
[root@master1 ~]# kubectl apply -f service_nodeport.yaml
[root@master1 ~]# kubectl get svc -l run=my-nginx-nodeport
NAME                TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)
my-nginx-nodeport   NodePort   10.100.156.7   <none>        80:30380/TCP
[root@master1 ~]# curl 10.100.156.7
<!DOCTYPE html>
<h1>Welcome to nginx!</h1>
```

```
</html>
```

注意：10.100.156.7 是 k8s 集群内部的 service ip 地址，只能在 k8s 集群内部访问，在集群外无法访问。

```
[root@master1 ~]# curl 192.168.40.180:30380     //在集群外访问 service
<!DOCTYPE html>
<h1>Welcome to nginx!</h1>
</html>
```

在浏览器访问 service：


```
[root@master1 ~]# ipvsadm -Ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  172.17.0.1:30380 rr
  -> 10.244.102.86:80             Masq    1      0          0
  -> 10.244.121.36:80             Masq    1      0          0
```

转发流程：

客户端请求 http://192.168.40.180:30380 -> docker0 虚拟网卡:172.17.0.1:30380 -> 10.244.121.36/86:80

11.5 创建 Service：type 类型是 ExternalName

应用场景：跨名称空间访问

需求：default 名称空间下的 client 服务想要访问 nginx-ns 名称空间下的 nginx-svc 服务。

```
[root@master1 ~]# vim server_nginx.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: nginx-ns
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
```

```
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        imagePullPolicy: IfNotPresent
[root@master1 ~]# kubectl create ns nginx-ns
[root@master1 ~]# kubectl apply -f server_nginx.yaml
[root@master1 ~]# kubectl get pods -n nginx-ns
nginx-7cf7d6dbc8-lzm6j   1/1     Running   0          10m

[root@master1 ~]# vim nginx_svc.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-svc
  namespace: nginx-ns
spec:
  selector:
    app: nginx
  ports:
   - name: http
     protocol: TCP
     port: 80
     targetPort: 80
[root@master1 ~]# kubectl apply -f nginx_svc.yaml

[root@master1 ~]# vim client.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: client
spec:
  replicas: 1
  selector:
    matchLabels:
      app: busybox
  template:
    metadata:
      labels:
        app: busybox
    spec:
      containers:
      - name: busybox
```

```
        image: busybox
        imagePullPolicy: IfNotPresent
        command: ["/bin/sh","-c","sleep 36000"]
[root@master1 ~]# kubectl apply -f client.yaml

[root@master1 ~]# vim client_svc.yaml
apiVersion: v1
kind: Service
metadata:
  name: client-svc
spec:
  type: ExternalName
  externalName: nginx-svc.nginx-ns.svc.cluster.local
  ports:
  - name: http
    port: 80
    targetPort: 80
[root@master1 ~]# kubectl get pods
client-76b6556d97-xk7mg              1/1     Running      0
[root@master1 ~]# kubectl apply -f client_svc.yaml
```

该文件中指定了到 nginx-svc 的软链，让使用者感觉就好像调用自己命名空间的服务一样。

```
[root@master1 ~]# kubectl exec -it  client-76b6556d97-xk7mg -- /bin/sh    //登录到 client pod
/ # wget -q -O - client-svc.default.svc.cluster.local
    wget -q -O - nginx-svc.nginx-ns.svc.cluster.local    //两个请求的结果一样
```

11.6 k8s 最佳实践

11.6.1 k8s 集群引用外部的 mysql 数据库

在 node2 上安装 mysql 数据库：

```
[root@node2 ~]# yum install mariadb-server.x86_64 -y
[root@node2 ~]# systemctl start mariadb
[root@master1 ~]# vim mysql_service.yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  type: ClusterIP
  ports:
  - port: 3306
[root@master1 ~]# kubectl apply -f mysql_service.yaml
[root@master1 ~]# kubectl get svc | grep mysql
mysql               ClusterIP      10.107.232.103             3306/TCP
```

```
[root@master1 ~]# kubectl describe svc mysql
......
Port:              <unset>  3306/TCP
TargetPort:        3306/TCP
Endpoints:         <none>  #还没有 endpoint
Session Affinity:  None
......

[root@master1 ~]# vim mysql_endpoint.yaml
apiVersion: v1
kind: Endpoints
metadata:
  name: mysql
subsets:
- addresses:
  - ip: 192.168.40.182
  ports:
  - port: 3306
[root@master1 ~]# kubectl apply -f mysql_endpoint.yaml
[root@master1 ~]# kubectl describe svc mysql
......
Port:              <unset>  3306/TCP
TargetPort:        3306/TCP
Endpoints:         192.168.1.62:3306     #这个就是定义的外部数据库
......
```

上面配置就是将外部 IP 地址和服务引入到 k8s 集群内部，由 service 作为一个代理来达到能够访问外部服务的目的。

11.7 coredns 组件详解

11.7.1 DNS 是什么?

DNS 全称是 Domain Name System：域名系统，是整个互联网的电话簿，它能够将可被人理解的域名翻译成可被机器理解 IP 地址，使得互联网的使用者不再需要直接接触很难阅读和理解的 IP 地址。域名系统在现在的互联网中非常重要，因为服务器的 IP 地址可能会经常变动，如果没有了 DNS，那么可能 IP 地址一旦发生了更改，当前服务器的客户端就没有办法连接到目标的服务器了，如果我们为 IP 地址提供一个『别名』并在其发生变动时修改别名和 IP 地址的关系，那么我们就可以保证集群对外提供的服务能够相对稳定地被其他客户端访问。DNS 其实就是一个分布式的树状命名系统，它就像一个去中心化的分布式数据库，存储着从域名到 IP 地址的映射。

CoreDNS 其实就是一个 DNS 服务，而 DNS 作为一种常见的服务发现手段，所以很多开源项目以及工程师都会使用 CoreDNS 为集群提供服务发现的功能，Kubernetes 就在集群中使用 CoreDNS 解决服务发现的问题。 作为一个加入 CNCF（Cloud Native Computing Foundation）的服务，CoreDNS 的实现非常简单。

11.7.2 验证 coredns

```
[root@master1 ~]# vim dig.yaml
apiVersion: v1
kind: Pod
metadata:
  name: dig
```

```
  namespace: default
spec:
  containers:
  - name: dig
    image: busybox:1.28
    imagePullPolicy: IfNotPresent
    command:
      - sleep
      - "3600"
  restartPolicy: Always
[root@master1 ~]# kubectl apply -f dig.yaml
[root@master1 ~]# kubectl get svc | grep kubernetes
kubernetes          ClusterIP      10.96.0.1        443/TCP        5d13h
[root@master1 ~]# kubectl exec -it dig -- nslookup kubernetes  //解析 dns，如有以下返回说明 dns 安装成功
Server:         10.96.0.10
Address:  10.96.0.10#53
Name:kubernetes.default.svc.cluster.local
Address: 10.96.0.1

kubernetes.default.svc.cluster.local     服务名.名称空间.默认后缀
```

在 k8s 中创建 service 之后，service 默认的 FQDN 是<servicename>.<namespace>.svc.cluster.local，那么 k8s 集群内部的服务就可以通过 FQDN 访问。

12 K8s 持久化存储

在 k8s 中为什么要做持久化存储？

在 k8s 中部署的应用都是以 pod 容器的形式运行的，假如我们部署 MySQL、Redis 等数据库，需要对这些数据库产生的数据做备份。因为 Pod 是有生命周期的，如果 pod 不挂载数据卷，那 pod 被删除或重启后这些数据会随之消失，如果想要长久的保留这些数据就要用到 pod 数据持久化存储。

12.1 k8s 持久化存储：emptyDir

12.1.1 volume 参数详解

```
[root@master1 ~]# kubectl explain pods.spec.volumes    //查看 k8s 支持哪些存储
KIND:     Pod
VERSION:  v1
RESOURCE: volumes <[]Object>
FIELDS:
   awsElasticBlockStore <Object>
   azureDisk   <Object>
   azureFile   <Object>
   cephfs <Object>
   cinder <Object>
   configMap   <Object>
   csi    <Object>
```

```
   downwardAPI <Object>
   emptyDir    <Object>
   ephemeral   <Object>
   fc<Object>
   flexVolume  <Object>
   flocker     <Object>
   gcePersistentDisk    <Object>
   gitRepo     <Object>
   glusterfs   <Object>
   hostPath    <Object>
   iscsi  <Object>
   name   <string> -required-
   nfs    <Object>
   persistentVolumeClaim     <Object>
   photonPersistentDisk <Object>
   portworxVolume   <Object>
   projected   <Object>
   quobyte     <Object>
   rbd    <Object>
   scaleIO     <Object>
   secret <Object>
   storageos   <Object>
   vsphereVolume    <Object>
```

我们想要使用存储卷，需要经历如下步骤：

1、定义 pod 的 volume，这个 volume 指明它要关联到哪个存储上的。

2、在容器中要使用 volumemounts 挂载对应的存储。

emptyDir 类型的 Volume 是在 Pod 分配到 Node 上时被创建，Kubernetes 会在 Node 上自动分配一个目录，因此无需指定宿主机 Node 上对应的目录文件。 这个目录的初始内容为空，当 Pod 从 Node 上移除时，emptyDir 中的数据会被永久删除。emptyDir Volume 主要用于某些应用程序无需永久保存的临时目录，多个容器的共享目录等。

Emptydir 的官方网址：https://kubernetes.io/docs/concepts/storage/volumes#emptydir

除了必需的 `path` 属性之外，你可以选择性地为 `hostPath` 卷指定 `type`。

支持的 `type` 值如下：

取值	行为
	空字符串（默认）用于向后兼容，这意味着在安装 hostPath 卷之前不会执行任何检查。
`DirectoryOrCreate`	如果在给定路径上什么都不存在，那么将根据需要创建空目录，权限设置为 0755，具有与 kubelet 相同的组和属主信息。
`Directory`	在给定路径上必须存在的目录。
`FileOrCreate`	如果在给定路径上什么都不存在，那么将在那里根据需要创建空文件，权限设置为 0644，具有与 kubelet 相同的组和所有权。
`File`	在给定路径上必须存在的文件。
`Socket`	在给定路径上必须存在的 UNIX 套接字。
`CharDevice`	在给定路径上必须存在的字符设备。
`BlockDevice`	在给定路径上必须存在的块设备。

12.1.2 创建一个 pod，挂载临时目录 emptyDir

```
[root@master1 ~]# vim emptydir.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-empty
spec:
  containers:
  - name: container-empty
    image: nginx
    imagePullPolicy: IfNotPresent
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - emptyDir: {}
    name: cache-volume
[root@master1 ~]# kubectl apply -f emptydir.yaml  //更新资源清单文件
[root@master1 ~]# kubectl get pods -o wide | grep empty  //查看 pod 调度到哪个节点
pod-empty      1/1     Running     0       10.244.209.152   node1
[root@master1 ~]# kubectl get pods pod-empty -o yaml | grep uid  //查看 pod 的 uid
uid: 38d60544-8591-468d-b70d-2a66df3a1cf6
```

登录到 node1 上：

```
[root@node1 ~]# yum -y install tree
[root@node1 ~]# tree /var/lib/kubelet/pods/38d60544-8591-468d-b70d-2a66df3a1cf6
/var/lib/kubelet/pods/38d60544-8591-468d-b70d-2a66df3a1cf6
├── containers
│   └── container-empty
│       └── c9e06d01
├── etc-hosts
├── plugins
│   └── kubernetes.io~empty-dir
│       ├── cache-volume
│       │   └── ready
│       └── wrapped_default-token-cq5qp
│           └── ready
└── volumes
    ├── kubernetes.io~empty-dir
    │   └── cache-volume
    │       └── aa
    └── kubernetes.io~secret
        └── default-token-cq5qp
            ├── ca.crt -> ..data/ca.crt
```

```
├── namespace -> ..data/namespace
└── token -> ..data/token
```

由上可知，临时目录在本地的 /var/lib/kubelet/pods/38d60544-8591-468d-b70d-2a66df3a1cf6/volumes/kubernetes.io~empty-dir/cache-volume/ 下

12.2 k8s 持久化存储：hostPath

hostPath Volume 是指 Pod 挂载宿主机上的目录或文件。 hostPath Volume 使得容器可以使用宿主机的文件系统进行存储，hostpath（宿主机路径）：节点级别的存储卷，在 pod 被删除，这个存储卷还是存在的，不会被删除，所以只要同一个 pod 被调度到同一个节点上来，在 pod 被删除重新被调度到这个节点之后，对应的数据依然是存在的。

12.2.1 查看 hostPath 存储卷的用法

```
[root@master1 ~]# kubectl explain pods.spec.volumes.hostPath
KIND:     Pod
VERSION:  v1
RESOURCE: hostPath <Object>
FIELDS:
   path   <string> -required-
   type   <string>
```

12.2.2 创建一个 pod，挂载 hostPath 存储卷

```
[root@master1 ~]# vim hostpath.yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-hostpath
spec:
  containers:
  - image: nginx
    imagePullPolicy: IfNotPresent
    name: test-nginx
    volumeMounts:
    - mountPath: /test-nginx
      name: test-volume
  - image: tomcat:8.5-jre8-alpine
    imagePullPolicy: IfNotPresent
    name: test-tomcat
    volumeMounts:
    - mountPath: /test-tomcat
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      path: /data1
      type: DirectoryOrCreate
```

注意：DirectoryOrCreate 表示本地有/data1 目录，就用本地的，本地没有就会在 pod 调度到的节点自动创建一个。

https://kubernetes.io/zh-cn/docs/concepts/storage/volumes/

```
[root@master1 ~]# kubectl apply -f hostpath.yaml
[root@master1 ~]# kubectl get pods -o wide | grep hostpath
test-hostpath       2/2     Running         10.244.209.153   node1
```

由上面可以知道 pod 调度到了 node1 上，登录到 node1 机器，查看是否在这台机器创建了存储目录。

```
[root@node1 ~]# ll /data1/  #上面可以看到已经创建了存储目录/data1，这个/data1 会作为 pod 的持久化存储目录。
[root@node1 ~]# cd /data1/  #在 node1 上的/data1 下创建一个目录：
[root@node1 data1]# mkdir aa
```

测试存储卷是否可以正常使用,登录到 nginx 容器：

```
[root@master1 ~]# kubectl exec -it test-hostpath -c test-nginx -- /bin/bash
root@test-hostpath:/# ls /test-nginx/    #/test-nginx/目录存在，说明已经把宿主机目录挂载到了容器里
```

测试存储卷是否可以正常使用,登录到 tomcat 容器：

```
[root@master1 ~]# kubectl exec -it test-hostpath -c test-tomcat -- /bin/bash
root@test-hostpath:/# ls /test-tomcat/    #/test-tomcat/目录存在，说明已经成功挂载
```

通过上面测试可以看到，同一个 pod 里的 test-nginx 和 test-tomcat 这两个容器是共享存储卷的。

hostpath 存储卷缺点：单节点，pod 删除之后重新创建必须调度到同一个 node 节点，数据才不会丢失。

可以用分布式存储：nfs，cephfs，glusterfs。

12.3 k8s 持久化存储：nfs

NFS：网络文件系统，英文 Network File System(NFS)，是由 SUN 公司研制的 UNIX 表示层协议(presentation layer protocol)，能使使用者访问网络上别处的文件就像在使用自己的计算机一样。

12.3.1 搭建 nfs 服务

以 k8s 的控制节点作为 NFS 服务端：

```
[root@master1 ~]# yum install nfs-utils -y
[root@master1 ~]# mkdir /data/volumes -pv   //在宿主机创建 NFS 需要的共享目录
[root@master1 ~]# systemctl start nfs
[root@master1 ~]# vim /etc/exports  //配置 nfs 共享服务器上的/data/volumes 目录
/data/volumes *(rw,no_root_squash)
# rw 该主机对该共享目录有读写权限
# no_root_squash 登入 NFS 主机使用分享目录的使用者，如果是 root 的话，那么对于这个分享的目录来说，他就具有 root 的权限。
[root@master1 ~]# exportfs -arv  //使 NFS 配置生效
[root@master1 ~]# ststemctl restart nfs
[root@master1 ~]# systemctl enable nfs  //设置成开机自启动
[root@master1 ~]# systemctl status nfs  //查看 nfs 是否启动成功
```

node2 和 node1 上也安装 nfs 驱动：

```
# yum -y install nfs-utils
# systemctl enable nfs --now
```

在 node1 上手动挂载试试：

```
[root@node1 ~]# mkdir /test
```

```
[root@node1 ~]# mount 192.168.40.180:/data/volumes /test/
[root@node1 ~]# df -h
192.168.40.180:/data/volumes  50G  5.2G   45G  11% /test
[root@node1 ~]# umount /test  //手动卸载
```

12.3.2 创建 Pod，挂载 NFS 共享出来的目录

Pod 挂载 nfs 的官方地址：https://kubernetes.io/zh/docs/concepts/storage/volumes/

```
[root@master1 ~]# vim nfs.yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-nfs-volume
spec:
  containers:
  - name: test-nfs
    image: nginx
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 80
      protocol: TCP
    volumeMounts:
    - name: nfs-volumes
      mountPath: /usr/share/nginx/html
  volumes:
  - name: nfs-volumes
    nfs:
      path: /data/volumes            #nfs 的共享目录
      server: 192.168.40.180         #master1 机器的 ip，这个是安装 nfs 服务的地址
[root@master1 ~]# kubectl apply -f nfs.yaml
[root@master1 ~]# kubectl get pods -o wide
nfs-test-76ffd8fdd6-5cbsk   1/1     Running   0          36s   10.244.121.44   node1
```

登录到 nfs 服务器，在共享目录创建一个 index.html

```
[root@master1 ~]# cd /data/volumes/
[root@master1 volumes]# vim index.html
Hello, everyone!
[root@master1]# curl 10.244.102.94  //请求 pod，看结果
Hello, everyone!
```

通过上面可以看到，在共享目录创建的 index.html 已经被 pod 挂载了。

登录到 pod 验证下：

```
[root@master1 ~]# kubectl exec -it nfs-test-76ffd8fdd6-5cbsk -- /bin/bash
root@test-nfs-volume:/# cat /usr/share/nginx/html/index.html
Hello, everyone!
```

上面说明挂载 nfs 存储卷成功了，nfs 支持多个客户端挂载，可以创建多个 pod，挂载同一个 nfs 服务器共享出来的目录；但是 nfs 如果宕机了，数据也就丢失了，所以需要使用分布式存储，常见的分布式存储有 glusterfs 和 cephfs。

12.4 k8s 持久化存储：PVC

参考官网:https://kubernetes.io/docs/concepts/storage/persistent-volumes/#access-modes

12.4.1 k8s PV 是什么？

PersistentVolume（PV）是群集中的一块存储，由管理员配置或使用存储类动态配置。它是集群中的资源，就像 pod 是 k8s 集群资源一样。PV 是容量插件，如 Volumes，其生命周期独立于使用 PV 的任何单个 pod。

12.4.2 k8s PVC 是什么？

PersistentVolumeClaim（PVC）是一个持久化存储卷，我们在创建 pod 时可以定义这个类型的存储卷。它类似于一个 pod。Pod 消耗节点资源，PVC 消耗 PV 资源。 Pod 可以请求特定级别的资源（CPU 和内存）。pvc 在申请 pv 的时候也可以请求特定的大小和访问模式（例如，可以一次读写或多次只读）。

12.4.3 k8s PVC 和 PV 工作原理

PV 是群集中的资源。 PVC 是对这些资源的请求。

PV 和 PVC 之间的相互作用遵循以下生命周期：

（1）pv 的供应方式：可以通过两种方式配置 PV，静态或动态。

静态的：集群管理员创建了许多 PV。它们包含可供群集用户使用的实际存储的详细信息。它们存在于 Kubernetes API 中，可供使用。

动态的：当管理员创建的静态 PV 都不匹配用户的 PersistentVolumeClaim 时，群集可能会尝试为 PVC 专门动态配置卷。此配置基于 StorageClasses，PVC 必须请求存储类，管理员必须创建并配置该类，以便进行动态配置。

（2）绑定：用户创建 pvc 并指定需要的资源和访问模式。在找到可用 pv 之前，pvc 会保持未绑定状态。

（3）使用：

a）需要找一个存储服务器，把它划分成多个存储空间；

b）k8s 管理员可以把这些存储空间定义成多个 pv；

c）在 pod 中使用 pvc 类型的存储卷之前需要先创建 pvc，通过定义需要使用的 pv 的大小和对应的访问模式，找到合适的 pv；

d）pvc 被创建之后，就可以当成存储卷来使用了，我们在定义 pod 时就可以使用这个 pvc 的存储卷；

e）pvc 和 pv 它们是一一对应的关系，pv 如果被 pvc 绑定了，就不能被其他 pvc 使用了；

f）我们在创建 pvc 的时候，应该确保和底下的 pv 能绑定，如果没有合适的 pv，那么 pvc 就会处于 pending 状态。

（4）回收策略：当我们创建 pod 时如果使用 pvc 做为存储卷，那么它会和 pv 绑定，当删除 pod，pvc 和 pv 绑定就会解除，解除之后和 pvc 绑定的 pv 卷里的数据需要怎么处理，目前，卷可以保留，回收或删除：

Recycle （不推荐使用，1.15 可能被废弃了）

Retain：当删除 pvc 的时候，pv 仍然存在，处于 released 状态，但是它不能被其他 pvc 绑定使用，里面的数据还是存在的，当我们下次再使用的时候，数据还是存在的，这个是默认的回收策略。

Delete：删除 pvc 时即会从 Kubernetes 中移除 PV，也会从相关的外部设施中删除存储资产。

12.4.4 创建 pod，使用 pvc 作为持久化存储卷

创建 nfs 共享目录：

```
[root@master1 ~]# mkdir /data/volume_test/v{1..4} -p  //在宿主机创建 NFS 需要的共享目录
[root@master1 ~]# vim /etc/exports  //配置 nfs 共享宿主机上的/data/volume_test/v1..v10 目录
/data/volumes *(rw,no_root_squash)
/data/volume_test/v1 *(rw,no_root_squash)
/data/volume_test/v2 *(rw,no_root_squash)
/data/volume_test/v3 *(rw,no_root_squash)
/data/volume_test/v4 *(rw,no_root_squash)
[root@master1 ~]# exportfs -arv  //重新加载配置，使配置成效
```

```
[root@master1 ~]# kubectl explain pv  //查看定义 pv 需要的字段
KIND:     PersistentVolume
VERSION:  v1
FIELDS:
   apiVersion <string>s
   kind <string>
   metadata <Object>
   spec <Object>
[root@master1 ~]# kubectl explain pv.spec.nfs  //查看定义 nfs 类型的 pv 需要的字段
KIND:     PersistentVolume
VERSION:  v1
RESOURCE: nfs <Object>
FIELDS:
   path   <string> -required-
   readOnly    <boolean>
   server <string> -required-
```

创建 pv：参考 https://kubernetes.io/zh/docs/concepts/storage/persistent-volumes/#reclaiming

```
[root@master1 ~]# vim pv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: v1
  labels:
    app: v1
spec:
  nfs:
    server: 192.168.40.180
    path: /data/volume_test/v1
  accessModes: ["ReadWriteOnce"]
  capacity:
    storage: 1Gi
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: v2
  labels:
    app: v2
spec:
  nfs:
    server: 192.168.40.180
    path: /data/volume_test/v2
  accessModes: ["ReadOnlyMany"]
```

```
  capacity:
    storage: 2Gi
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: v3
  labels:
    app: v3
spec:
  nfs:
    server: 192.168.40.180
    path: /data/volume_test/v3
  accessModes: ["ReadWriteMany"]
  capacity:
    storage: 3Gi
[root@master1 ~]# kubectl apply -f pv.yaml
[root@master1 ~]# kubectl get pv
NAME   CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS   REASON   AGE
v1     1Gi        RWO            Retain           Available                                   55s
v2     2Gi        ROX            Retain           Available                                   55s
v3     3Gi        RWX            Retain           Available                                   55s
```

注解：　RWO：readwariteonce：　单路读写，允许同一个 node 节点上的 pod 访问

　　ROX：readonlymany：　多路只读，允许不通 node 节点的 pod 以只读方式访问

　　RWX：readwritemany：　多路读写，允许不同的 node 节点的 pod 以读写方式访问

创建 pvc，和符合条件的 pv 绑定：

```
[root@master1 ~]# vim pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-v1
spec:
  accessModes: ["ReadWriteOnce"]
  selector:
    matchLabels:
      app: v1
  resources:
    requests:
      storage: 1Gi
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-v2
```

```
spec:
  accessModes: ["ReadOnlyMany"]
  selector:
    matchLabels:
      app: v2
  resources:
    requests:
      storage: 2Gi
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-v3
spec:
  accessModes: ["ReadWriteMany"]
  selector:
    matchLabels:
      app: v3
  resources:
    requests:
      storage: 3Gi
[root@master1]# kubectl apply -f pvc.yaml
[root@master1]# kubectl get pvc
NAME     STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvc-v1   Bound    v1       1Gi        RWO                           17s
pvc-v2   Bound    v2       2Gi        ROX                           17s
pvc-v3   Bound    v3       3Gi        RWX                           17s
```

看到 pvc 的 status 都是 bound 状态，就说明 pvc 跟 pv 已经绑定了。

创建 pod，挂载 pvc：

```
[root@master1 ~]# vim pod_pvc.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pvc-test
spec:
  replicas: 3
  selector:
    matchLabels:
      cunchu: pvc
  template:
    metadata:
      labels:
        cunchu: pvc
```

```
  spec:
    containers:
    - name: test-pvc
      image: nginx
      imagePullPolicy: IfNotPresent
      ports:
      - containerPort: 80
        protocol: TCP
      volumeMounts:
      - name: nginx-html
        mountPath: /usr/share/nginx/html
    volumes:
    - persistentVolumeClaim:
        claimName: pvc-v1
      name: nginx-html
[root@master1 ~]# kubectl apply -f pod_pvc.yaml
[root@master1 ~]# kubectl get pods -l cunchu=pvc
NAME                        READY   STATUS    RESTARTS   AGE
pvc-test-65b6487489-84tlz   1/1     Running   0          6m51s
pvc-test-65b6487489-9kdgd   1/1     Running   0          6m51s
pvc-test-65b6487489-xpb7n   1/1     Running   0          6m51s
[root@master1 ~]# cd /data/volume_test/v1
[root@master1 v1]# mkdir lucky
[root@master1]# kubectl exec -it pvc-test-65b6487489-84tlz -- ls /usr/share/nginx/html/
lucky
```

注：使用 pvc 和 pv 的注意事项

1、我们每次创建 pvc 的时候，需要事先有划分好的 pv，这样可能不方便，那么可以在创建 pvc 的时候直接动态创建一个 pv 这个存储类，pv 事先是不存在的。

2、pvc 和 pv 绑定，如果使用默认的回收策略 retain，那么删除 pvc 之后，pv 会处于 released 状态，我们想要继续使用这个 pv，需要手动删除 pv，kubectl delete pv pv_name，删除 pv，不会删除 pv 里的数据，当我们重新创建 pvc 时还会和这个最匹配的 pv 绑定，数据还是原来数据，不会丢失。

经过测试，如果回收策略是 Delete，删除 pv，pv 后端存储的数据也不会被删除。

回收策略：persistentVolumeReclaimPolicy 字段

删除 pvc 的步骤：需要先删除使用 pvc 的 pod，再删除 pvc。

```
[root@master1 ~]# kubectl delete -f pod_pvc.yaml
[root@master1 ~]# kubectl get pvc
NAME     STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvc-v1   Bound    v1       1Gi        RWO                           36s
pvc-v2   Bound    v2       2Gi        ROX                           22m
pvc-v3   Bound    v3       3Gi        RWX                           22m
[root@master1 ~]# kubectl delete pvc pvc-v1
```

演示 pv 用 Delete 回收策略：

```
[root@master1 ~]# vim pv-1.yaml
```

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: v4
  labels:
    app: v4
spec:
  nfs:
    server: 192.168.40.180
    path: /data/volume_test/v4
  accessModes: ["ReadWriteOnce"]
  capacity:
    storage: 1Gi
  persistentVolumeReclaimPolicy: Delete
[root@master1 ~]# kubectl apply -f pv-1.yaml
[root@master1 ~]# vim pvc-1.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-v4
spec:
  accessModes: ["ReadWriteOnce"]
  selector:
    matchLabels:
      app: v4
  resources:
    requests:
      storage: 1Gi
[root@master1 ~]# kubectl apply -f pvc-1.yaml
[root@master1 ~]# kubectl get pvc
NAME     STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvc-v1   Bound    v1       1Gi        RWO                           5m4s
pvc-v2   Bound    v2       2Gi        ROX                           28m
pvc-v3   Bound    v3       3Gi        RWX                           28m
pvc-v4   Bound    v4       1Gi        RWO                           33s
[root@master1 ~]# vim pod_pvc-2.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pvc-test-1
spec:
  replicas: 3
  selector:
    matchLabels:
```

```
      cunchu: pvc-1
  template:
    metadata:
      labels:
        cunchu: pvc-1
    spec:
      containers:
      - name: test-pvc
        image: nginx
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
          protocol: TCP
        volumeMounts:
        - name: nginx-html
          mountPath: /usr/share/nginx/html
      volumes:
      - persistentVolumeClaim:
          claimName: pvc-v4
        name: nginx-html
[root@master1 ~]# kubectl apply -f pod_pvc-2.yaml
[root@master1 ~]# cd /data/volume_test/v4/
[root@master1 v4]# mkdir lucky
[root@master1 ~]# kubectl exec -it pvc-test-1-744c9dbc96-5b6c2 -- ls /usr/share/nginx/html/
lucky
```

12.5 k8s 存储类：storageclass

上面介绍的 PV 和 PVC 模式都是需要先创建好 PV，然后定义好 PVC 和 pv 进行一对一的 Bond，但是如果 PVC 请求成千上万，那么就需要创建成千上万的 PV，对于运维人员来说维护成本很高，Kubernetes 提供一种自动创建 PV 的机制，叫 StorageClass，它的作用就是创建 PV 的模板。k8s 集群管理员通过创建 storageclass 可以动态生成一个存储卷 pv 供 k8s pvc 使用。

每个 StorageClass 都包含字段 provisioner，parameters 和 reclaimPolicy。

具体来说，StorageClass 会定义以下两部分：

1、PV 的属性，比如存储的大小、类型等；

2、创建这种 PV 需要使用到的存储插件，比如 Ceph、NFS 等；

有了这两部分信息，Kubernetes 就能够根据用户提交的 PVC，找到对应的 StorageClass，然后 Kubernetes 就会调用 StorageClass 声明的存储插件，创建出需要的 PV。

12.5.1 查看定义的 storageclass 需要的字段

```
[root@master1 ~]# kubectl explain storageclass
KIND:     StorageClass
VERSION:  storage.k8s.io/v1
FIELDS:
   allowVolumeExpansion <boolean>
   allowedTopologies    <[]Object>
```

apiVersion <string>

kind <string>

metadata <Object>

mountOptions <[]string>

parameters <map[string]string>

provisioner <string> -required-

reclaimPolicy <string>

volumeBindingMode <string>

provisioner：供应商，storageclass 需要有一个供应者，用来确定我们使用什么样的存储来创建 pv，常见的 provisioner 如下：https://kubernetes.io/zh/docs/concepts/storage/storage-classes/

Volume Plugin	Internal Provisioner	Config Example
AWSElasticBlockStore	✓	AWS EBS
AzureFile	✓	Azure File
AzureDisk	✓	Azure Disk
CephFS	-	-
Cinder	✓	OpenStack Cinder
FC	-	-
Flexvolume	-	-
Flocker	✓	-
GCEPersistentDisk	✓	GCE PD
Glusterfs	✓	Glusterfs
iSCSI	-	-
Quobyte	✓	Quobyte
NFS	-	-
RBD	✓	Ceph RBD
VsphereVolume	✓	vSphere
PortworxVolume	✓	Portworx Volume
ScaleIO	✓	ScaleIO
StorageOS	✓	StorageOS
Local	-	Local

provisioner 既可以由内部供应商提供，也可以由外部供应商提供，如果是外部供应商可以参考 https://github.com/kubernetes-incubator/external-storage/ 下提供的方法创建。

以 NFS 为例，要想使用 NFS，我们需要一个 nfs-client 的自动装载程序，称之为 provisioner，这个程序会使用我们已经配置好的 NFS 服务器自动创建持久卷，也就是自动帮我们创建 PV。

reclaimPolicy：回收策略

allowVolumeExpansion：允许卷扩展，PersistentVolume 可以配置成可扩展。将此功能设置为 true 时，允许用户通过编辑相应的 PVC 对象来调整卷大小。当基础存储类的 allowVolumeExpansion 字段设置为 true 时，以下类型的卷支持卷扩展。

卷类型	Kubernetes 版本要求
gcePersistentDisk	1.11
awsElasticBlockStore	1.11
Cinder	1.11
glusterfs	1.11
rbd	1.11
Azure File	1.11
Azure Disk	1.11
Portworx	1.11
FlexVolume	1.13
CSI	1.14 (alpha), 1.16 (beta)

注意：此功能仅用于扩容卷，不能用于缩小卷。

12.5.2 安装 nfs provisioner，用于配合存储类动态生成 pv

把 nfs-subdir-external-provisioner.tar.gz 上传到 node1 和 node2 上，手动解压。

```
[root@node1 ~]# ctr -n=k8s.io images import nfs-subdir-external-provisioner.tar.gz
[root@node2 ~]# ctr -n=k8s.io images import nfs-subdir-external-provisioner.tar.gz
```

创建运行 nfs-provisioner 需要的 sa 账号：

```
[root@master1 ~]# vim serviceaccount.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: nfs-provisioner
[root@master1 ~]# kubectl apply -f serviceaccount.yaml
```

什么是 sa？sa 的全称是 serviceaccount，是为了方便 Pod 里面的进程调用 Kubernetes API 或其他外部服务而设计的。

指定 serviceaccount 之后，我们把 pod 创建出来，我们在使用这个 pod 时，这个 pod 就有了我们指定的账户的权限了。

对 sa 授权：

```
[root@master1 ~]# kubectl create clusterrolebinding nfs-provisioner-clusterrolebinding --
clusterrole=cluster-admin --serviceaccount=default:nfs-provisioner
```

安装 nfs-provisioner 程序

```
[root@master1 ~]# mkdir /data/nfs_pro -p
[root@master1 ~]# vim /etc/exports   //把/data/nfs_pro 变成 nfs 共享的目录
/data/nfs_pro *(rw,no_root_squash)
[root@master1 ~]# exportfs -arv
exporting *:/data/nfs_pro
[root@master1 ~]# vim nfs-deployment.yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nfs-provisioner
spec:
  selector:
    matchLabels:
       app: nfs-provisioner
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: nfs-provisioner
    spec:
      serviceAccount: nfs-provisioner
      containers:
        - name: nfs-provisioner
          image: registry.cn-beijing.aliyuncs.com/mydlq/nfs-subdir-external-provisioner:v4.0.0
          imagePullPolicy: IfNotPresent
```

```
            volumeMounts:
              - name: nfs-client-root
                mountPath: /persistentvolumes
            env:
              - name: PROVISIONER_NAME
                value: example.com/nfs
              - name: NFS_SERVER
                value: 192.168.40.180
              - name: NFS_PATH
                value: /data/nfs_pro
        volumes:
          - name: nfs-client-root
            nfs:
              server: 192.168.40.180
              path: /data/nfs_pro
[root@master1 ~]# kubectl apply -f nfs-deployment.yaml
[root@master1 ~]# kubectl get pods | grep nfs
nfs-provisioner-cd5589cfc-pjwsq      1/1     Running
```

12.5.3 创建 storageclass，动态供给 pv

```
[root@master1 ~]# vim nfs-storageclass.yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: nfs
provisioner: example.com/nfs
[root@master1 ~]# kubectl apply -f nfs-storageclass.yaml
[root@master1 ~]# kubectl get storageclass
NAME   PROVISIONER       RECLAIMPOLICY   VOLUMEBINDINGMODE
nfs    example.com/nfs   Delete          Immediate
```

注：provisioner 处写的 example.com/nfs 应该跟安装 nfs provisioner 时候的 env 下的 PROVISIONER_NAME 的 value 值保持一致，如下：

```
env:
- name: PROVISIONER_NAME
  value: example.com/nfs
```

12.5.4 创建 pvc，通过 storageclass 动态生成 pv

```
[root@master1 ~]# vim claim.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: test-claim1
spec:
  accessModes: ["ReadWriteMany"]
  resources:
    requests:
```

```
      storage: 1Gi
  storageClassName: nfs
[root@master1 ~]# kubectl apply -f claim.yaml
[root@master1 ~]# kubectl get pvc  //查看是否动态生成了 pv，pvc 是否创建成功，并和 pv 绑定
```

通过上面可以看到 test-claim1 的 pvc 已经成功创建了，这个 pvc 是由 storageclass 调用 nfs provisioner 自动生成。

步骤总结：

1、供应商：创建一个 nfs provisioner

2、创建 storageclass，storageclass 指定刚才创建的供应商

3、创建 pvc，这个 pvc 指定 storageclass

12.5.5 创建 pod，挂载 storageclass 动态生成的 pvc：test-claim1

```
[root@master1 ~]# vim read-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: read-pod
spec:
  containers:
  - name: read-pod
    image: nginx
    imagePullPolicy: IfNotPresent
    volumeMounts:
      - name: nfs-pvc
        mountPath: /usr/share/nginx/html
  restartPolicy: "Never"
  volumes:
    - name: nfs-pvc
      persistentVolumeClaim:
        claimName: test-claim1
[root@master1 ~]# kubectl apply -f read-pod.yaml
[root@master1 ~]# kubectl get pods | grep read
read-pod      1/1     Running     0
```

13 Statefulset 控制器

13.1 概念、原理解读

StatefulSet 是为了管理有状态服务的问题而设计的。

有状态服务：StatefulSet 是有状态的集合，管理有状态的服务，它所管理的 Pod 的名称不能随意变化。数据持久化的目录也是不一样，每一个 Pod 都有自己独有的数据持久化存储目录。比如 MySQL 主从、redis 集群等。

无状态服务：RC、Deployment、DaemonSet 都是管理无状态的服务，它们所管理的 Pod 的 IP、名字，启停顺序等都是随机的。个体对整体无影响，所有 pod 都是共用一个数据卷的，部署的 tomcat 就是无状态的服务，tomcat 被删除，在启动一个新的 tomcat，加入到集群即可，跟 tomcat 的名字无关。

13.2 Statefulset 资源清单文件编写技巧

13.2.1 查看定义 Statefulset 资源需要的字段

```
[root@master1 ~]# kubectl explain statefulset
KIND:     StatefulSet
VERSION:  apps/v1
FIELDS:
   apiVersion <string> #定义 statefulset 资源需要使用的 api 版本
   kind   <string>          #定义的资源类型
   metadata    <Object>     #元数据
   spec   <Object>          #定义容器相关的信息
```

13.2.2 查看 statefulset.spec 字段如何定义

```
[root@master1 ~]# kubectl explain statefulset.spec
KIND:     StatefulSet
VERSION:  apps/v1
RESOURCE: spec <Object>
FIELDS:
   podManagementPolicy  <string> #pod 管理策略
   replicas    <integer>  #副本数
   revisionHistoryLimit <integer> #保留的历史版本
   selector    <Object> -required- #标签选择器，选择它所关联的 pod
   serviceName <string> -required-  #headless service 的名字
   template    <Object> -required-     #生成 pod 的模板
   updateStrategy  <Object>   #更新策略
   volumeClaimTemplates <[]Object> #存储卷申请模板
```

13.2.3 查看 statefulset 的 spec.template 字段如何定义

```
[root@master1 ~]# kubectl explain statefulset.spec.template
KIND:     StatefulSet
VERSION:  apps/v1
RESOURCE: template <Object>
FIELDS:
   metadata    <Object>
   spec   <Object>  #定义容器属性的
```

通过上面可以看到，statefulset 资源中有两个 spec 字段。第一个 spec 声明的是 statefulset 定义多少个 Pod 副本（默认将仅部署 1 个 Pod）、匹配 Pod 标签的选择器、创建 pod 的模板、存储卷申请模板，第二个 spec 是 spec.template.spec：主要用于 Pod 里的容器属性等配置。

13.3 Statefulset 使用案例

13.3.1 部署 web 站点

```
[root@master1 ~]# vim statefulset.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
```

```
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx
  serviceName: "nginx"
  replicas: 2
  template:
    metadata:
     labels:
       app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: ["ReadWriteOnce"]
      storageClassName: "nfs"
      resources:
        requests:
          storage: 1Gi
[root@master1 ~]# kubectl apply -f statefulset.yaml
[root@master1 ~]# kubectl get statefulset
```

```
NAME    READY    AGE
web       2/2       42s
[root@master1 ~]# kubectl get pods -l app=nginx
web-0    1/1      Running    0          2m17s
web-1    1/1      Running    0          115s
```

通过上面可以看到创建的 pod 是有序的。

```
[root@master1 ~]# kubectl get svc -l app=nginx
NAME     TYPE         CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
nginx    ClusterIP    None          <none>         80/TCP     3m19s
[root@master1 ~]# kubectl get pvc
NAME       STATUS  VOLUME                                     CAPACITY    ACCESS MODES    STORAGECLASS   AGE
www-web-0 Bound   pvc-39a9755f-3248-49ff-8f9e-5b068b609c8f 1Gi        RWO,RWX         nfs-web        7m45s
www-web-1 Bound   pvc-be93d4a3-1aca-44cc-802f-ddeb38c05018 1Gi        RWO,RWX         nfs-web        7m41s
[root@master1 ~]# kubectl get pv
NAME  CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM         STORAGECLASS   REASON   AGE
pvc-39a9755f-3248-49ff-8f9e-5b068b609c8f  1Gi  RWO,RWX Delete Bound default/www-web-0   nfs-web   8m3s
pvc-be93d4a3-1aca-44cc-802f-ddeb38c05018  1Gi  RWO,RWX Delete Bound default/www-web-1   nfs-web   7m59s
```

StatefulSet 由以下几个部分组成：

1. Headless Service：用来定义 pod 网路标识，生成可解析的 DNS 记录
2. volumeClaimTemplates：存储卷申请模板，创建 pvc，指定 pvc 名称大小，自动创建 pvc，且 pvc 由存储类供应。
3. StatefulSet：管理 pod 的

扩展：什么是 Headless service?

Headless service 不分配 clusterIP，headless service 可以通过解析 service 的 DNS,返回所有 Pod 的 dns 和 ip 地址(statefulSet 部署的 Pod 才有 DNS)，普通的 service,只能通过解析 service 的 DNS 返回 service 的 ClusterIP。

1.headless service 会为 service 分配一个域名

<service name>.$<namespace name>.svc.cluster.local

K8s 中资源的全局 FQDN 格式:

Service_NAME.NameSpace_NAME.Domain.LTD.

Domain.LTD.=svc.cluster.local. #这是默认 k8s 集群的域名。

FQDN 全称 Fully Qualified Domain Name，即全限定域名：同时带有主机名和域名的名称。

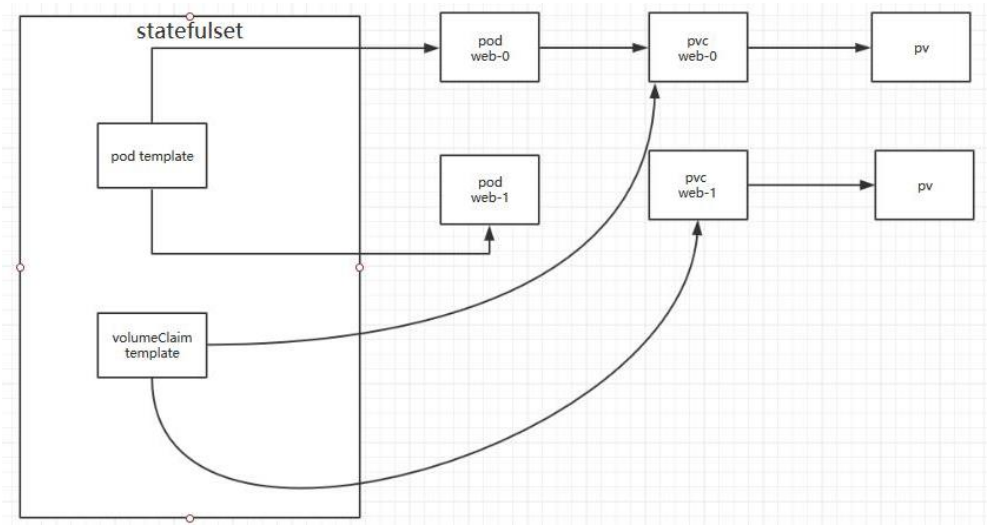
2. StatefulSet 会为关联的 Pod 保持一个不变的 Pod Name

statefulset 中 Pod 的名字格式为$(StatefulSet name)-$(pod 序号)

3. StatefulSet 会为关联的 Pod 分配一个 dnsName

$<Pod Name>.$<service name>.$<namespace name>.svc.cluster.local

为什么要用 volumeClaimTemplate：对于有状态应用都会用到持久化存储，比如 mysql 主从，由于主从数据库的数据是不能存放在一个目录下的，每个 mysql 节点都需要有自己独立的存储空间。而在 deployment 中创建的存储卷是一个共享的存储卷，多个 pod 使用同一个存储卷，它们数据是同步的，而 statefulset 定义中的每一个 pod 都不能使用同一个存储卷，这就需要使用 volumeClainTemplate，当在使用 statefulset 创建 pod 时，volumeClainTemplate 会自动生成一个 PVC，从而请求绑定一个 PV，每一个 pod 都有自己专用的存储卷。Pod、PVC 和 PV 对应的关系图如下：



使用kubectl run运行一个提供nslookup命令的容器的,这个命令来自于dnsutils包,通过对pod主机名执行nslookup,可以检查它们在集群内部的DNS地址:

```
[root@master1 ~]# kubectl run busybox --image docker.io/library/busybox:1.28  --image-pull-policy=IfNotPresent --restart=Never --rm -it busybox -- sh
root@web-1:/# nslookup web-0.nginx.default.svc.cluster.local  #statefulset 创建的 pod 也是有 dns 记录的
Name:      web-0.nginx.default.svc.cluster.local
Address 1: 10.244.104.62 web-0.nginx.default.svc.cluster.local

root@web-1:/# nslookup nginx.default.svc.cluster.local  #查询 service dns，会把对应的 pod ip 解析出来
Name:      nginx.default.svc.cluster.local
Address 1: 10.244.166.146 web-1.nginx.default.svc.cluster.local
Address 2: 10.244.104.62 web-0.nginx.default.svc.cluster.local

root@web-1:/# dig -t A nginx.default.svc.cluster.local @10.96.0.10
; <<>> DiG 9.11.5-P4-5.1+deb10u3-Debian <<>> -t A nginx.default.svc.cluster.local @10.96.0.10
......
;; ANSWER SECTION:
nginx.default.svc.cluster.local. 30 IN     A     10.244.209.139
nginx.default.svc.cluster.local. 30 IN     A     10.244.209.140
```

扩展：举例说明service 和headless service区别

1、通过deployment创建pod，pod前端创建一个service

```
[root@master1 ~]# vim deploy-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  type: ClusterIP
```

```
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    run: my-nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: busybox
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
        command:
          - sleep
          - "3600"
[root@master1 ~]# kubectl apply -f deploy-service.yaml
[root@master1 ~]# kubectl get svc -l run=my-nginx
NAME       TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
my-nginx   ClusterIP   10.100.89.90   <none>        80/TCP
[root@master1 ~]# kubectl get pods -l run=my-nginx
NAME                        READY   STATUS    RESTARTS   AGE
my-nginx-58f74fc5b6-jzbvk   1/1     Running   0          70s
my-nginx-58f74fc5b6-n9lqv   1/1     Running   0          53s
```

通过上面可以看到 deployment 创建的 pod 是随机生成的。

```
[root@master1 ~]# kubectl exec -it web-1 -- /bin/bash   //进入到 web-1 的 pod
root@web-1:/# nslookup my-nginx.default.svc.cluster.local
Name:my-nginx.default.svc.cluster.local
Address: 10.100.89.90   #解析的是 service 的 ip 地址
```

Statefulset 总结：

1、Statefulset 管理的 pod，pod 名字是有序的，由 statefulset 的名字-0、1、2 这种格式组成。

2、创建 statefulset 资源的时候，必须事先创建好一个 service,如果创建的 service 没有 ip，那对这个 service 做 dns 解析，会找到它所关联的 pod ip，如果创建的 service 有 ip，那对这个 service 做 dns 解析，会解析到 service 本身 ip。

3、statefulset 管理的 pod，删除 pod，新创建的 pod 名字跟删除的 pod 名字是一样的。

4、statefulset 具有 volumeclaimtemplate 这个字段，这个是卷申请模板，会自动创建 pv，pvc 也会自动生成，跟 pv 进行绑定，那如果创建的 statefulset 使用了 volumeclaimtemplate 这个字段，那创建 pod，数据目录是独享的。

5、ststefulset 创建的 pod，是域名的（域名组成：pod-name.svc-name.svc-namespace.svc.cluster.local）。

13.4 Statefulset 管理 pod

13.4.1 Statefulset 实现 pod 的动态扩容

如果我们觉得两个副本太少了,想要增加,只需要修改配置文件statefulset.yaml里的replicas的值即可,原来replicas: 2，现在变成 replicaset: 3，修改之后，执行如下命令更新：

```
[root@master1 ~]# kubectl apply -f statefulset.yaml
[root@master1 ~]# kubectl get sts
NAME   READY   AGE
web    3/3     60m
[root@master1 ~]# kubectl get pods -l app=nginx
NAME    READY   STATUS    RESTARTS   AGE
web-0   1/1     Running   0          61m
web-1   1/1     Running   0          60m
web-2   1/1     Running   0          79s
```

也可以直接编辑控制器实现扩容：

```
[root@master1 ~]# kubectl edit sts web
```

把 spec 下的 replicas 后面的值改成 4，保存退出。

```
[root@master1 ~]# kubectl get pods -l app=nginx
NAME    READY   STATUS    RESTARTS   AGE
web-0   1/1     Running   0          62m
web-1   1/1     Running   0          62m
web-2   1/1     Running   0          3m13s
web-3   1/1     Running   0          26s
```

13.4.2 Statefulset 实现 pod 的动态缩容

如果我们觉得 4 个 Pod 副本太多了，想要减少，只需要修改配置文件 statefulset.yaml 里的 replicas 的值即可，把 replicaset：4 变成 replicas: 2，修改之后，执行如下命令更新：

```
[root@master1 ~]# kubectl apply -f statefulset.yaml
[root@master1 ~]# kubectl get pods -l app=nginx
NAME    READY   STATUS    RESTARTS   AGE
web-0   1/1     Running   0          64m
web-1   1/1     Running   0          64m
```

13.4.3 Statefulset 实现 pod 的更新

```
[root@master1 ~]# kubectl explain sts.spec.updateStrategy
```

```
[root@master1 ~]# vim statefulset.yaml
......
spec:
  updateStrategy:
    rollingUpdate:
      partition: 1
      maxUnavailable: 0
......
containers:
- name: nginx
  image: ikubernetes/myapp:v2
  imagePullPolicy: IfNotPresent
.....
```

在一个终端动态查看 pod:

```
[root@master1 ~]# kubectl get pods -l app=nginx -w
```

另一个终端执行如下命令：

```
[root@master1 ~]# kubectl apply -f statefulset.yaml
```

出现的结果如下：

```
web-0                              1/1     Running   0          10m
web-1                              1/1     Running   0          10m
web-1                              1/1     Terminating   0          10m
web-1                              1/1     Terminating   0          10m
web-1                              0/1     Terminating   0          10m
web-1                              0/1     Terminating   0          10m
web-1                              0/1     Terminating   0          10m
web-1                              0/1     Pending       0          0s
web-1                              0/1     Pending       0          0s
web-1                              0/1     ContainerCreating   0          0s
web-1                              0/1     ContainerCreating   0          1s
web-1                              1/1     Running             0          2s
web-1                              1/1     Running             0          11s
```

从上面结果可以看出来，pod 在更新的时候，只是更新了 web-1 这个 pod， partition: 1 表示更新的时候会把 pod 序号大于等于 1 的进行更新。

如果更新策略是 OnDelete，那不会自动更新 pod，需要手动删除，重新常见的 pod 才会实现更新。

```
spec:
  updateStrategy:
    type: OnDelete
  replicas: 2
```

14 DaemonSet 控制器

14.1 DaemonSet 概述

DaemonSet 控制器能够确保 k8s 集群所有的节点都运行一个相同的 pod 副本，当向 k8s 集群中增加 node 节点时，这个

node 节点也会自动创建一个 pod 副本，当 node 节点从集群移除，这些 pod 也会自动删除；删除 Daemonset 也会删除它们创建的 pod。

14.1.1 DaemonSet 工作原理

daemonset 的控制器会监听 kuberntes 的 daemonset 对象、pod 对象、node 对象，这些被监听的对象之变动，就会触发 syncLoop 循环让 kubernetes 集群朝着 daemonset 对象描述的状态进行演进。

14.1.2 Daemonset 典型的应用场景

在集群的每个节点上运行存储，比如：glusterd 或 ceph。

在每个节点上运行日志收集组件，比如：flunentd、logstash、filebeat 等。

在每个节点上运行监控组件，比如：Prometheus、Node Exporter、collectd 等。

14.1.3 DaemonSet 与 Deployment 的区别

Deployment 部署的副本 Pod 会分布在各个 Node 上，每个 Node 都可能运行好几个副本。

DaemonSet 的不同之处在于：每个 Node 上最多只能运行一个副本。

14.2 DaemonSet 资源清单文件编写技巧

14.2.1 查看定义 Daemonset 资源需要的字段有哪些?

```
[root@master1 ~]# kubectl explain ds
KIND:     DaemonSet
VERSION:  apps/v1
FIELDS:
   apiVersion <string>   #当前资源使用的 api 版本，跟 VERSION:  apps/v1 保持一致
   kind   <string>       #资源类型，跟 KIND:  DaemonSet 保持一致
   metadata    <Object>  #元数据，定义 DaemonSet 名字的
   spec   <Object>       #定义容器的
   status <Object>       #状态信息，不能改
```

14.2.2 查看 DaemonSet 的 spec 字段如何定义?

```
[root@master1 ~]# kubectl explain ds.spec
KIND:     DaemonSet
VERSION:  apps/v1
RESOURCE: spec <Object>
FIELDS:
   minReadySeconds <integer>   #当新的 pod 启动几秒种后，再 kill 掉旧的 pod。
   revisionHistoryLimit <integer> #历史版本
   selector    <Object> -required-  #用于匹配 pod 的标签选择器
   template    <Object> -required- #定义 Pod 的模板，基于这个模板定义的所有 pod 是一样的
   updateStrategy  <Object> #daemonset 的升级策略
```

14.2.3 查看 DaemonSet 的 spec.template 字段如何定义?

```
[root@master1 ~]# kubectl explain ds.spec.template
KIND:     DaemonSet
VERSION:  apps/v1
RESOURCE: template <Object>
FIELDS:
   metadata    <Object>
   spec<Object>
```

14.3 DaemonSet 使用案例

14.3.1 部署日志收集组件 fluentd

```
[root@master1 ~]# vim daemonset.yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      - key: node-role.kubernetes.io/control-plane
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: fluentd:latest
        imagePullPolicy: IfNotPresent
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
```

```
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
[root@master1 ~]# kubectl apply -f daemonset.yaml
[root@master1 ~]# kubectl get ds -n kube-system
NAME                    DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE
fluentd-elasticsearch   3         3         3       3            3
[root@master1 ~]# kubectl get pods -n kube-system -o wide
```

通过上面可以看到在 k8s 的三个节点均创建了 fluentd 这个 pod，pod 的名字是由控制器的名字-随机数组成的。

14.4 Daemonset 管理 pod

14.4.1 DaemonSet 实现 pod 的滚动更新

```
[root@master1 ~]# kubectl explain ds.spec.updateStrategy     //查看 daemonset 的滚动更新策略
KIND:     DaemonSet
VERSION:  apps/v1
RESOURCE: updateStrategy <Object>
FIELDS:
   rollingUpdate   <Object>
   type   <string>
[root@master1 ~]# kubectl explain ds.spec.updateStrategy.rollingUpdate     //查看支持的更新策略
KIND:     DaemonSet
VERSION:  apps/v1
RESOURCE: rollingUpdate <Object>
FIELDS:
   maxUnavailable  <string>
```

上面表示 rollingUpdate 更新策略只支持 maxUnavailabe，先删除在更新；因为我们不支持一个节点运行两个 pod，因此需要先删除一个，在更新一个。

更新镜像版本，可以按照如下方法：

```
[root@master1 ~]# kubectl set image daemonsets fluentd-elasticsearch fluentd-elasticsearch=ikubernetes/filebeat:5.6.6-alpine -n kube-system
```

15 Configmap

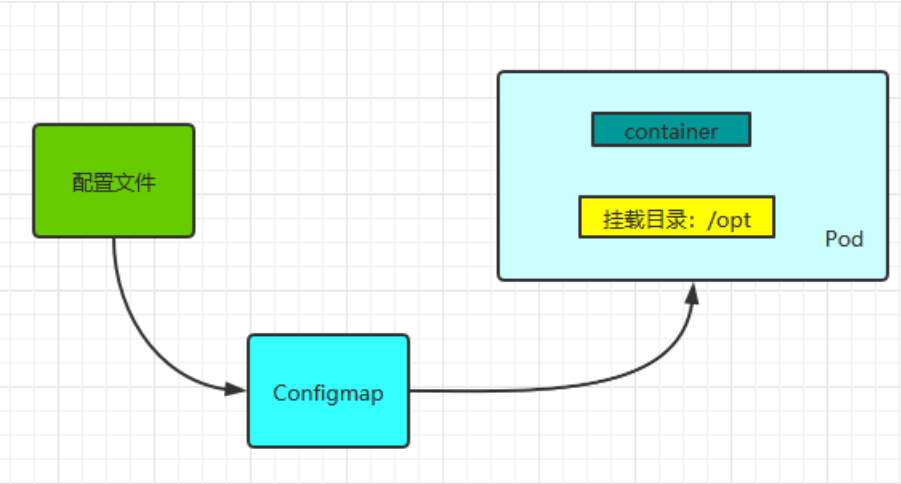
15.1 Configmap 概述

15.1.1 什么是 Configmap?

Configmap 是 k8s 中的资源对象，用于保存非机密性的配置的，数据可以用 key/value 键值对的形式保存，也可通过文件的形式保存。

15.1.2 Configmap 能解决哪些问题?

我们在部署服务的时候，每个服务都有自己的配置文件，如果一台服务器上部署多个服务：nginx、tomcat、apache 等，那么这些配置都存在这个节点上，假如一台服务器不能满足线上高并发的要求，需要对服务器扩容，扩容之后的服务器还是需要部署多个服务：nginx、tomcat、apache，新增加的服务器上还是要管理这些服务的配置，如果有一个服务出现问题，需要修改配置文件，每台物理节点上的配置都需要修改，这种方式肯定满足不了线上大批量的配置变更要求。 所以，k8s 中引入了 Configmap 资源对象，可以当成 volume 挂载到 pod 中，实现统一的配置管理。



1、Configmap 是 k8s 中的资源，相当于配置文件，可以有一个或者多个 Configmap；

2、Configmap 可以做成 Volume，k8s pod 启动之后，通过 volume 形式映射到容器内部指定目录上；

3、容器中应用程序按照原有方式读取容器特定目录上的配置文件；

4、在容器看来，配置文件就像是打包在容器内部特定目录，整个过程对应用没有任何侵入。

15.1.3 Configmap 应用场景

1、使用 k8s 部署应用，当你将应用配置写进代码中，更新配置时也需要打包镜像，configmap 可以将配置信息和 docker 镜像解耦，以便实现镜像的可移植性和可复用性，因为一个 configMap 其实就是一系列配置信息的集合，可直接注入到 Pod 中给容器使用。configmap 注入方式有两种，一种将 configMap 做为存储卷，一种是将 configMap 通过 env 中 configMapKeyRef 注入到容器中。

2、使用微服务架构的话，存在多个服务共用配置的情况，如果每个服务中单独一份配置的话，那么更新配置就很麻烦，使用 configmap 可以友好的进行配置共享。

15.1.4 局限性

ConfigMap 在设计上不是用来保存大量数据的。在 ConfigMap 中保存的数据不可超过 1 MiB。如果你需要保存超出此尺寸限制的数据，可以考虑挂载存储卷或者使用独立的数据库或者文件服务。

15.2 Configmap 创建方法

15.2.1 命令行直接创建

直接在命令行中指定 configmap 参数创建，通过--from-literal 指定参数

```
[root@master1 ~]# kubectl create configmap tomcat-config --from-literal=tomcat_port=8080 --from-literal=server_name=myapp.tomcat.com
[root@master1 ~]# kubectl describe configmap tomcat-config
Name:         tomcat-config
Namespace:    default
Labels:       <none>
Annotations:  <none>
Data
====
server_name:
----
myapp.tomcat.com
tomcat_port:
----
```

```
8080
Events:  <none>
```

15.2.2 通过文件创建

通过指定文件创建一个 configmap，--from-file=<文件>

```
[root@master1 ~]# vim nginx.conf
server {
  server_name www.nginx.com;
  listen 80;
  root /home/nginx/www/
}
```

定义一个 key 是 www，值是 nginx.conf 中的内容：

```
[root@master1 ~]# kubectl create configmap www-nginx --from-file=www=./nginx.conf
[root@master1 ~]# kubectl describe configmap www-nginx
Name:         www-nginx
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
www:
----
server {
  server_name www.nginx.com;
  listen 80;
  root /home/nginx/www/
}
```

15.2.3 指定目录创建 configmap

```
[root@master1 ~]# mkdir test
[root@master1 ~]# cd test/
[root@master1 test]# vim my-server.cnf
server-id=1
[root@master1 test]# vim my-slave.cnf
server-id=2
```

指定目录创建 configmap：

```
[root@master1 ~]# kubectl create configmap mysql-config --from-file=/root/test/
[root@master1 ~]# kubectl describe configmap mysql-config    //查看 configmap 详细信息
Name:         mysql-config
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
```

```
my-server.cnf:
----
server-id=1
my-slave.cnf:
----
server-id=2
Events:  <none>
```

15.2.4 编写 configmap 资源清单 YAML 文件

```
[root@master1 ~]# vim mysql-configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql
  labels:
    app: mysql
data:
  master.cnf: |
    [mysqld]
    log-bin
    log_bin_trust_function_creators=1
    lower_case_table_names=1
  slave.cnf: |
    [mysqld]
    super-read-only
    log_bin_trust_function_creators=1
```

15.3 使用 Configmap

15.3.1 通过环境变量引入：使用 configMapKeyRef

创建一个存储 mysql 配置的 configmap:

```
[root@master1 ~]# vim mysql-configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql
  labels:
    app: mysql
data:
  log: "1"
  lower: "1"
[root@master1 ~]# kubectl apply -f mysql-configmap.yaml
[root@master1 ~]# vim mysql-pod.yaml   //创建 pod，引用 Configmap 中的内容:
apiVersion: v1
kind: Pod
metadata:
```

```
  name: mysql-pod
spec:
  containers:
  - name: mysql
    image: busybox
    command: [ "/bin/sh", "-c", "sleep 3600" ]
    env:
    - name: log_bin       #定义环境变量 log_bin
      valueFrom:
        configMapKeyRef:
          name: mysql     #指定 configmap 的名字
          key: log        #指定 configmap 中的 key
    - name: lower         #定义环境变量 lower
      valueFrom:
        configMapKeyRef:
          name: mysql
          key: lower
  restartPolicy: Never
[root@master1 ~]# kubectl apply -f mysql-pod.yaml
[root@master1 ~]# kubectl exec -it mysql-pod -- /bin/sh
/ # printenv
log_bin=1
lower=1
```

15.3.2 通过环境变量引入：使用 envfrom

```
[root@master1 ~]# vim mysql-pod-envfrom.yaml
apiVersion: v1
kind: Pod
metadata:
  name: mysql-pod-envfrom
spec:
  containers:
  - name: mysql
    image: busybox
    imagePullPolicy: IfNotPresent
    command: [ "/bin/sh", "-c", "sleep 3600" ]
    envFrom:
    - configMapRef:
        name: mysql     #指定 configmap 的名字
  restartPolicy: Never
[root@master1 ~]# kubectl apply -f mysql-pod-envfrom.yaml
[root@master1 ~]# kubectl exec -it mysql-pod-envfrom -- /bin/sh
/ # printenv
lower=1
log=1
```

15.3.3 把 configmap 做成 volume，挂载到 pod

```
[root@master1 ~]# vim mysql-configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql
  labels:
    app: mysql
data:
    log: "1"
    lower: "1"
    my.cnf: |
      [mysqld]
      Welcome=Welcome to mysql.
[root@master1 ~]# kubectl apply -f mysql-configmap.yaml

[root@master1 ~]# vim mysql-pod-volume.yaml
apiVersion: v1
kind: Pod
metadata:
  name: mysql-pod-volume
spec:
  containers:
  - name: mysql
    image: busybox
    command: [ "/bin/sh","-c","sleep 3600" ]
    volumeMounts:
    - name: mysql-config
      mountPath: /tmp/config
  volumes:
  - name: mysql-config
    configMap:
      name: mysql
  restartPolicy: Never
[root@master1 ~]# kubectl apply -f mysql-pod-volume.yaml
[root@master1 ~]# kubectl exec -it mysql-pod-volume -- ls /tmp/config/
log    lower   my.cnf
```

15.4 Configmap 热更新

```
[root@master1 ~]# kubectl edit configmap mysql
把 logs: “1”变成 log: “2”后保存退出。
[root@master1 ~]# kubectl exec -it mysql-pod-volume -- /bin/sh
/ # cat /tmp/config/log
2
```

注意：更新 ConfigMap 后：使用该 ConfigMap 挂载的 Env 不会同步更新；使用该 ConfigMap 挂载的 Volume 中的数据需要一段时间（实测大概 10 秒）才能同步更新。

16 Secret

16.1 Secret 是什么?

上面我们学习的 Configmap 一般是用来存放明文数据的，如配置文件，对于一些敏感数据，如密码、私钥等数据时，要用 secret 类型。

Secret 解决了密码、token、秘钥等敏感数据的配置问题，而不需要把这些敏感数据暴露到镜像或者 Pod Spec 中。Secret 可以以 Volume 或者环境变量的方式使用。

要使用 secret，pod 需要引用 secret。Pod 可以用两种方式使用 secret：作为 volume 中的文件被挂载到 pod 中的一个或者多个容器里，或者当 kubelet 为 pod 拉取镜像时使用。

secret 可选参数有三种：

generic：通用类型，通常用于存储密码数据。

tls：此类型仅用于存储私钥和证书。

docker-registry：若要保存 docker 仓库的认证信息的话，就必须使用此种类型来创建。

Secret 类型：

Service Account：用于被 serviceaccount 引用。serviceaccout 创建时 Kubernetes 会默认创建对应的 secret。Pod 如果使用了 serviceaccount，对应的 secret 会自动挂载到 Pod 的/run/secrets/kubernetes.io/serviceaccount 目录中。

Opaque：base64 编码格式的 Secret，用来存储密码、秘钥等。可以通过 base64 —decode 解码获得原始数据，安全性弱。

kubernetes.io/dockerconfigjson：用来存储私有 docker registry 的认证信息。

16.2 使用 Secret

16.2.1 通过环境变量引入 Secret

把 mysql 的 root 用户的 password 创建成 secret

```
[root@master1 ~]# kubectl create secret generic mysql-password --from-literal=password=pod**lucky66
[root@master1 ~]# kubectl get secret
NAME                  TYPE                                  DATA   AGE
mysql-password        Opaque                                1      30s
[root@master1 ~]# kubectl describe secret mysql-password
Name:         mysql-password
Namespace:    default
Labels:       <none>
Annotations:  <none>
Type:  Opaque
Data
====
password: 12bytes    #password 的值是加密的，但 secret 的加密是一种伪加密，它仅仅是将数据做了 base64 的编码。
```

16.2.2 创建 pod，引用 secret

```
[root@master1 ~]# vim pod-secret.yaml
apiVersion: v1
kind: Pod
metadata:
```

```
  name: pod-secret
  labels:
    app: myapp
spec:
  containers:
  - name: myapp
    image: ikubernetes/myapp:v1
    imagePullPolicy: IfNotPresent
    ports:
    - name: http
      containerPort: 80
    env:
    - name: MYSQL_ROOT_PASSWORD    #它是 Pod 启动成功后，Pod 中容器的环境变量名
      valueFrom:
        secretKeyRef:
          name: mysqlpassword      #这是 secret 的对象名
          key: password            #它是 secret 中的 key 名
[root@master1 ~]# kubectl apply -f pod-secret.yaml
[root@master1 ~]# kubectl exec -it pod-secret -- /bin/sh
/ # printenv
MYSQL_ROOT_PASSWORD=pod**lucky66
```

16.2.3 通过 volume 挂载 Secret

1）创建 Secret

```
[root@master1 ~]# echo -n 'admin' | base64    //手动加密，基于 base64 加密
YWRtaW4=
[root@master1 ~]# echo -n '123456' | base64
MTIzNDU2
[root@master1 ~]# echo MTIzNDU2 | base64 -d   //解码
```

2）创建 yaml 文件

```
[root@master1 ~]# vim secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MTIzNDU2
[root@master1 ~]# kubectl apply -f secret.yaml
[root@master1 ~]# kubectl describe secret mysecret
Name:         mysecret
Namespace:    default
Labels:       <none>
Annotations:  <none>
```

```
Type:  Opaque
Data
====
password:  6 bytes
username:  5 bytes
```

3）将 Secret 挂载到 Volume 中

```
[root@master1 ~]# vim pod_secret_volume.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-secret-volume
spec:
  containers:
  - name: myapp
    image: busybox:1.28
    imagePullPolicy: IfNotPresent
    command: ["/bin/sh","-c","sleep 3600"]
    volumeMounts:
    - name: secret-volume
      mountPath: /etc/secret
      readOnly: true
  volumes:
  - name: secret-volume
    secret:
      secretName: mysecret
[root@master1 ~]# kubectl apply -f pod_secret_volume.yaml
[root@master1 ~]# kubectl exec -it pod-secret-volume -- /bin/sh
/ # ls /etc/secret
password  username
/ # cat /etc/secret/username
admin
/ # cat /etc/secret/password
123456
```

由上可见，在 pod 中的 secret 信息实际已经被解密。

17 k8s 安全管理

17.1 安全管理

17.1.1 认证

认证基本介绍：

kubernetes 主要通过 APIserver 对外提供服务，那么就需要对访问 apiserver 的用户做认证，如果任何人都能访问 apiserver，那么就可以随意在 k8s 集群部署资源，这是非常危险的，也容易被黑客攻击渗透，所以需要我们对访问 k8s 系统

的 apiserver 的用户进行认证，确保是合法的符合要求的用户。

授权基本介绍：

认证通过后仅代表它是一个被 apiserver 信任的用户，能访问 apiserver，但是用户是否拥有删除资源的权限，需要进行授权操作，常见的授权方式有 rbac 授权。

准入控制基本介绍：

当用户经过认证和授权之后，最后一步就是准入控制了，k8s 提供了多种准入控制机制，它有点类似插件，为 apiserver 提供了很好的可扩展性。请求 apiserver 时，通过认证、鉴权后，持久化（api 对象保存到 etcd）前，会经过准入控制器，让它可以做变更和验证。

为什么需要准入控制器呢？

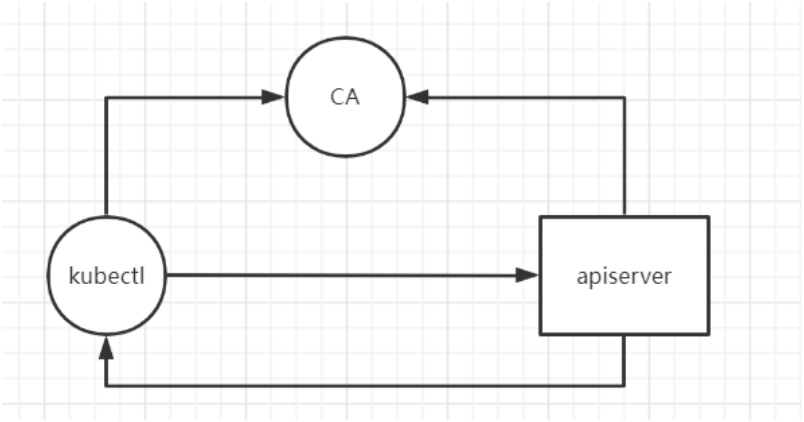
如果我们创建 pod 时定义了资源上下限，但不满足 LimitRange 规则中定义的资源上下限，此时 LimitRanger 就会拒绝我们创建此 pod。

假如我们定义了一个名称空间叫做 test，这个名称空间做下资源限制：限制最多可以使用 10vCPU、10Gi 内存，在这个名称空间 test 下创建的所有 pod，定义 limit 的时候，所有 pod 的 limit 值不能超过 test 这个名称空间设置的 limit 上线。

k8s 客户端访问 apiserver 的几种认证方式：

1）客户端认证：

客户端认证也称为双向 TLS 认证，kubectl 在访问 apiserver 的时候，apiserver 也要认证 kubectl 是否是合法的，他们都会通过 ca 根证书来进行验证，如下图：



2）Bearertoken：

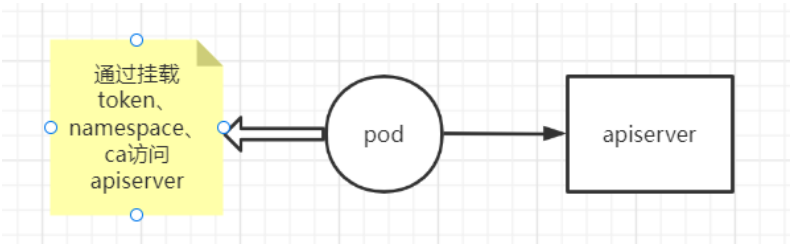
Bearertoken 的方式，可以理解为 apiserver 将一个密码通过了非对称加密的方式告诉了 kubectl，然后通过该密码进行相互访问，如下图：

Kubectl 访问 k8s 集群，要找一个 kubeconfig 文件，基于 kubeconfig 文件里的用户访问 apiserver

3）Serviceaccount

上面客户端证书认证和 Bearertoken 的两种认证方式，都是外部访问 apiserver 的时候使用的方式，那么我们这次说的 Serviceaccount 是内部访问 pod 和 apiserver 交互时候采用的一种方式。Serviceaccount 包括了，namespace、token、ca，且通过目录挂载的方式给予 pod，当 pod 运行起来的时候，就会读取到这些信息，从而使用该方式和 apiserver 进行通信。

如下图：



kubeconfig 文件官方地址：

https://kubernetes.io/zh-cn/docs/concepts/configuration/organize-cluster-access-kubeconfig/

在 K8S 集群当中，当我们使用 kubectl 操作 k8s 资源时候，需要确定我们用哪个用户访问哪个 k8s 集群，kubectl 操作 k8s 集群资源会去/root/.kube 目录下找 config 文件，可以通过 kubectl config 查看 config 文件配置，如下：

```
[root@master1 ~]# kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://192.168.40.180:6443          #apiserver 的地址
  name: kubernetes                               #集群的名字
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes              #上下文的名字
current-context: kubernetes-admin@kubernetes     #当前上下文的名字
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

在上面的配置文件当中，定义了集群、上下文以及用户。其中 Config 也是 K8S 的标准资源之一，在该配置文件当中定义了一个集群列表，指定的集群可以有多个；用户列表也可以有多个，指明集群中的用户；而在上下文列表当中，是进行定义可以使用哪个用户对哪个集群进行访问，以及当前使用的上下文是什么。

```
[root@master1 ~]# kubectl get pods --kubeconfig=/root/.kube/config
```

17.1.2 授权

用户通过认证之后，什么权限都没有，需要一些后续的授权操作，如对资源的增删该查等，kubernetes1.6 之后开始有 RBAC（基于角色的访问控制机制）授权检查机制。

Kubernetes 的授权是基于插件形成的，其常用的授权插件有以下几种：

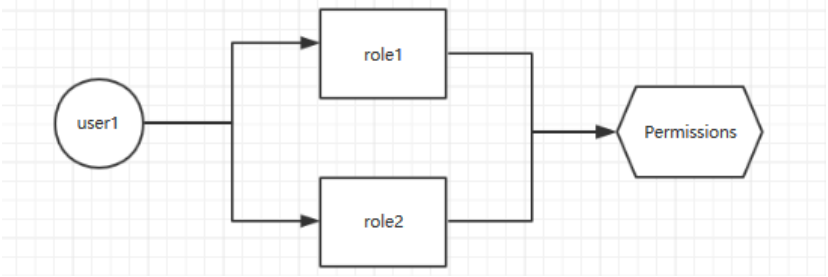
1）Node（节点认证）

2）ABAC（基于属性的访问控制）

3）RBAC（基于角色的访问控制）

4）Webhook（基于 http 回调机制的访问控制）

什么是 RBAC（基于角色的授权）：一个用户（Users）扮演一个角色（Role），角色拥有权限，从而让用户拥有这样的权限，随后在授权机制当中，只需要将权限授予某个角色，此时用户将获取对应角色的权限，从而实现角色的访问控制。如图：



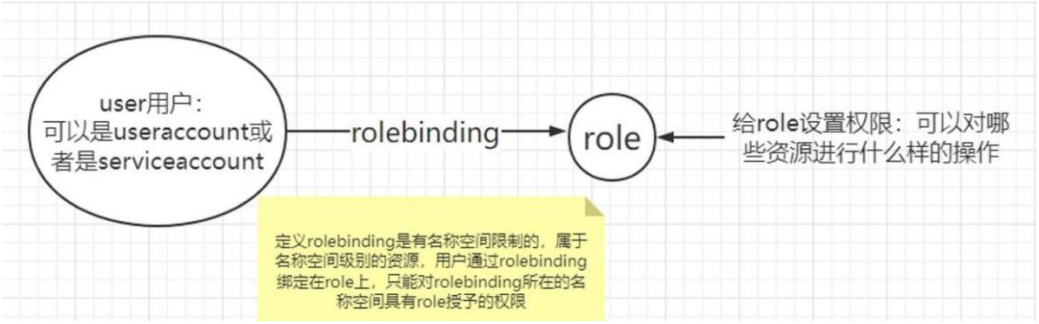
在 k8s 的授权机制当中，采用 RBAC 的方式进行授权，其工作逻辑是，把对对象的操作权限定义到一个角色当中，再将用户绑定到该角色，从而使用户得到对应角色的权限。如果通过 rolebinding 绑定 role，只能对 rolebingding 所在的名称空间的资源有权限，上图 user1 这个用户绑定到 role1 上，只对 role1 这个名称空间的资源有权限，对其他名称空间资源没有权限，属于名称空间级别的；

另外，k8s 为此还有一种集群级别的授权机制，就是定义一个集群角色（ClusterRole），对集群内的所有资源都有可操作的权限，从而将 User2 通过 ClusterRoleBinding 到 ClusterRole，从而使 User2 拥有集群的操作权限。

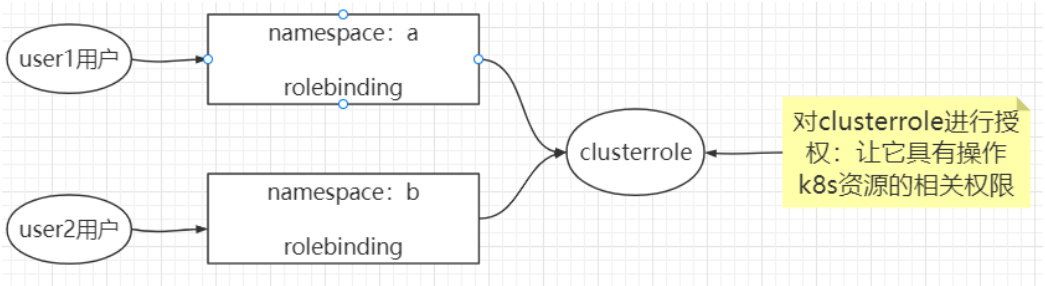
Role、RoleBinding、ClusterRole 和 ClusterRoleBinding 的关系如下：

1、用户基于 rolebinding 绑定到 role：

限定在 rolebinding 所在的名称空间。



2、用户基于 rolebinding 绑定到 clusterrole：



上面我们说了两个角色绑定：

（1）用户通过 rolebinding 绑定 role

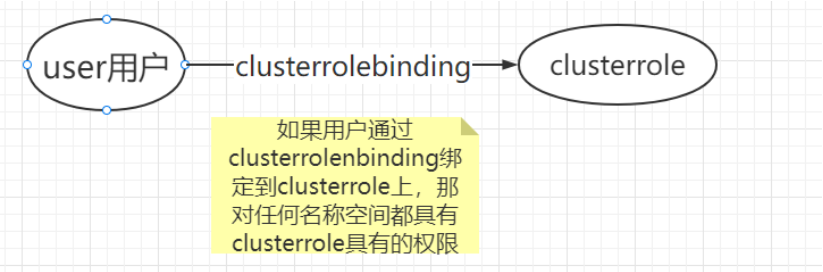
（2）用户通过 rolebinding 绑定 clusterrole

rolebinding 绑定 clusterrole 的好处：

假如有 6 个名称空间，每个名称空间的用户都需要对自己的名称空间有管理员权限，那么需要定义 6 个 role 和 rolebinding，然后依次绑定，如果名称空间更多，我们需要定义更多的 role，这个是很麻烦的，所以我们引入 clusterrole，定义一个 clusterrole，对 clusterrole 授予所有权限，然后用户通过 rolebinding 绑定到 clusterrole，就会拥有自己名称空间的管理员权限了。

注：RoleBinding 仅仅对当前名称空间有对应的权限。

3、用户基于 clusterrolebinding 绑定到 clusterrole：



用户基于 rbac 授权有几种方案：

基于 rolebinding 绑定到 role 上

基于 rolebinding 绑定到 clusterrole 上

基于 clusterrolebinding 绑定到 clusterrole 上

17.1.3 准入控制

在 k8s 上准入控制器的模块有很多，比较常用的有 LimitRanger、ResourceQuota、ServiceAccount、PodSecurityPolicy(k8s1.25 废弃了)等等，对于前面三种准入控制器系统默认是启用的，我们只需要定义对应的规则即可；对于 PodSecurityPolicy(k8s1.25 废弃了)这种准入控制器，系统默认没有启用，如果我们要使用，就必需启用以后，对应规则才会正常生效；这里需要注意一点，对应 psp 准入控制器，一定要先写好对应的规则，把规则和权限绑定好以后，在启用对应的准入控制器，否则先启用准入控制器，没有对应的规则，默认情况它是拒绝操作，这可能导致现有的 k8s 系统跑的系统级 pod 无法正常工作；所以对于 psp 准入控制器要慎用，如果规则和权限做的足够精细，它会给我们的 k8s 系统安全带来大幅度的提升，反之，可能导致整个 k8s 系统不可用。

```
[root@master1 ~]# vim /etc/kubernetes/manifests/kube-apiserver.yaml    //查看 apiserver 启用的准入控制器
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: 192.168.40.63:6443
  creationTimestamp: null
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --advertise-address=192.168.40.180
    ...
- --enable-admission-plugins=NodeRestriction
...
```

提示：apiserver 启用准入控制插件需要使用--enable-admission-plugins 选项来指定，该选项可以使用多个值，用逗号隔开表示启用指定的准入控制插件；这里配置文件中显式启用了 NodeRestrication 这个插件；默认没有写在这上面的内置准入控制器，它也是启用了的，比如 LimitRanger、ResourceQuota、ServiceAccount 等等；对于不同的 k8s 版本，内置的准入控制器和启用与否请查看相关版本的官方文档；对于那些没有启动的准入控制器，我们可以在上面选项中直接启用，分别

用逗号隔开即可。

对应的官网地址：https://kubernetes.io/zh-cn/docs/reference/access-authn-authz/admission-controllers/

17.2 Useraccount 和 ServiceAccount 介绍

17.2.1 Useraccount 和 ServiceAccount 概述

kubernetes 中账户分为：UserAccounts（用户账户）和 ServiceAccounts（服务账户）两种：

UserAccount 是给 kubernetes 集群外部用户使用的，如 kubectl 访问 k8s 集群要用 useraccount 用户， kubeadm 安装的 k8s，默认的 useraccount 用户是 kubernetes-admin。

k8s 客户端(一般用:kubectl) ---->API Server

APIServer 需要对客户端做认证，使用 kubeadm 安装的 K8s，会在用户家目录下创建一个认证配置文件 .kube/config 这里面保存了客户端访问 API Server 的密钥相关信息，这样当用 kubectl 访问 k8s 时，它就会自动读取该配置文件，向 API Server 发起认证，然后完成操作请求。

```
[root@master1~]# cat ~/.kube/config     //用户名称可以在 kubeconfig 中查看
...
users:
- name: kubernetes-admin
...
```

ServiceAccount 是 Pod 使用的账号，Pod 容器的进程需要访问 API Server 时用的就是 ServiceAccount 账户；ServiceAccount 仅局限它所在的 namespace，每个 namespace 创建时都会自动创建一个 default service account；创建 Pod 时，如果没有指定 Service Account，Pod 则会使用 default Service Account。

17.2.2 ServiceAccount 使用案例

1、创建 sa

```
[root@master1 ~]# kubectl create sa sa-test
```

2、创建 pod

```
[root@master1 ~]# vim pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: sa-test
  namespace: default
  labels:
    app: sa
spec:
  serviceAccountName: sa-test
  containers:
  - name: sa-tomcat
    ports:
    - containerPort: 80
    image: nginx
    imagePullPolicy: IfNotPresent
[root@master1 ~]# kubectl apply -f pod.yaml
[root@master1 ~]# kubectl exec -it sa-test -- /bin/bash
root@sa-test:/# cd /var/run/secrets/kubernetes.io/serviceaccount/
root@sa-test:/var/run/secrets/kubernetes.io/serviceaccount# curl --cacert ./ca.crt -H "Authorization:
```

Bearer $(cat ./token)" https://kubernetes/api/v1/namespaces/kube-system

显示如下：


```
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "namespaces \"kube-system\" is forbidden: User \"system:serviceaccount:default:sa-test\" cannot get resource \"namespaces\" in API group \"\" in the namespace \"kube-system\"",
  "reason": "Forbidden",
  "details": {
    "name": "kube-system",
    "kind": "namespaces"
  },
  "code": 403
}
```


上面结果能看到 sa 能通过 https 方式成功认证 API，但是没有权限访问 k8s 资源，所以 code 状态码是 403，表示没有权限操作 k8s 资源。

3、对 sa 做授权

[root@master1 ~]# kubectl create clusterrolebinding sa-test-admin --clusterrole=cluster-admin --serviceaccount=default:sa-test

4、再次请求

root@sa-test:/var/run/secrets/kubernetes.io/serviceaccount# curl --cacert ./ca.crt -H "Authorization: Bearer $(cat ./token)" https://kubernetes/api/v1/namespaces/kube-system

显示如下：


```
{
  "kind": "Namespace",
  "apiVersion": "v1",
  "metadata": {
    "name": "kube-system",
    "uid": "c392fcb2-12da-4419-b5de-6e2fe2ca626a",
    "resourceVersion": "28",
    "creationTimestamp": "2022-09-18T06:19:42Z",
    "labels": {
      "kubernetes.io/metadata.name": "kube-system"
    },
    "managedFields": [
      {
        "manager": "kube-apiserver",
        "operation": "Update",
        "apiVersion": "v1",
        "time": "2022-09-18T06:19:42Z",
        "fieldsType": "FieldsV1",
        "fieldsV1": {
```

```
            "f:metadata": {
              "f:labels": {
                ".": {},
                "f:kubernetes.io/metadata.name": {}
              }
            }
          }
        }
      ]
    },
    "spec": {
      "finalizers": [
        "kubernetes"
      ]
    },
    "status": {
      "phase": "Active"
    }
}
```


通过上面可以看到，对 sa 做授权之后就有权限访问 k8s 资源了。

root@sa-test:/var/run/secrets/kubernetes.io/serviceaccount# curl --cacert ./ca.crt -H "Authorization: Bearer $(cat ./token)" https://kubernetes/api/v1/pods

17.3 RBAC 认证授权策略

RBAC 介绍：在 Kubernetes 中，所有资源对象都是通过 API 进行操作，他们保存在 etcd 里。而对 etcd 的操作我们需要通过访问 kube-apiserver 来实现，上面的 Service Account 其实就是 APIServer 的认证过程，而授权的机制是通过 RBAC：基于角色的访问控制实现。

RBAC 有四个资源对象，分别是 Role、ClusterRole、RoleBinding、ClusterRoleBinding

17.3.1 Role：角色

一组权限的集合，在一个命名空间中，可以用其来定义一个角色，只能对命名空间内的资源进行授权。如果是集群级别的资源，则需要使用 ClusterRole。例如：定义一个角色用来读取 Pod 的权限。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: rbac
  name: pod-read
rules:
- apiGroups: [""]
  resources: ["pods"]
  resourceNames: []
  verbs: ["get","watch","list"]
```

rules 中的参数说明：

1、apiGroups：支持的 API 组列表，例如："apiVersion: apps/v1"等

2、resources：支持的资源对象列表，例如 pods、deployments、jobs 等

3、resourceNames: 指定 resource 的名称

4、verbs：对资源对象的操作方法列表。

17.3.2 ClusterRole：集群角色

具有和角色一致的命名空间资源的管理能力，还可用于以下特殊元素的授权

1、集群范围的资源，例如 Node

2、非资源型的路径，例如：/healthz

3、包含全部命名空间的资源，例如 Pods

例如：定义一个集群角色可让用户访问任意 secrets

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: secrets-clusterrole
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get","watch","list"]
```

17.3.3 RoleBinding：角色绑定、ClusterRolebinding：集群角色绑定

角色绑定和集群角色绑定用于把一个角色绑定在一个目标上，可以是 User，Group，Service Account，使用 RoleBinding 为某个命名空间授权，使用 ClusterRoleBinding 为集群范围内授权。

例如：将在 rbac 命名空间中把 pod-read 角色授予用户

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-read-bind
  namespace: rbac
subjects:
  kind: User
  name: es
  apiGroup: rbac.authorization.k8s.io
roleRef:
- kind: Role
  name: pod-read
  apiGroup: rbac.authorizatioin.k8s.io
```

RoleBinding 也可以引用 ClusterRole，对属于同一命名空间内的 ClusterRole 定义的资源主体进行授权，例如：es 能获取到集群中所有的资源信息。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: es-allresource
  namespace: rbac
subjects:
- kind: User
```

```
    name: es
    apiGroup: rbac.authorization.k8s.io
  roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: ClusterRole
    name: cluster-admin
```

17.4 资源的引用方式

多数资源可以用其名称的字符串表示，也就是 Endpoint 中的 URL 相对路径，例如 pod 中的日志是 GET /api/v1/namaspaces/{namespace}/pods/{podname}/log

如果需要在一个 RBAC 对象中体现上下级资源，就需要使用“/”分割资源和下级资源。

例如：若想授权让某个主体同时能够读取 Pod 和 Pod log，则可以配置 resources 为一个数组

```
[root@master1 ~]# kubectl create ns test
[root@master1 ~]# vim role-test.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: logs-reader
  namespace: test
rules:
- apiGroups: [""]
  resources: ["pods","pods/log"]
  verbs: ["get","list","watch"]
[root@master1 ~]# kubectl apply -f role-test.yaml
[root@master1 ~]# kubectl create sa sa-test -n test
[root@master1 ~]# kubectl create rolebinding sa-test-1 -n test --role=logs-reader --
serviceaccount=test:sa-test
[root@master1]# vim pod-test.yaml
apiVersion: v1
kind: Pod
metadata:
  name: sa-test-pod
  namespace: test
  labels:
    app: sa
spec:
  serviceAccountName: sa-test
  containers:
  - name:  sa-tomcat
    ports:
    - containerPort: 80
    image: nginx
    imagePullPolicy: IfNotPresent
[root@master1 ~]# kubectl apply -f pod-test.yaml
```

[root@master1 ~]# kubectl exec -it sa-test-pod -n test -- /bin/bash

root@sa-test-pod:/var/run/secrets/kubernetes.io/serviceaccount# curl --cacert ./ca.crt -H "Authorization: Bearer $(cat ./token)" https://kubernetes.default/api/v1/namespaces/test/pods/sa-test-pod/log

资源还可以通过名称（ResourceName）进行引用，在指定 ResourceName 后，使用 get、delete、update、patch 请求，就会被限制在这个资源实例范围内。

例如：下面的声明让一个主体只能对名为 my-configmap 的 Configmap 进行 get 和 update 操作：

```
apiVersion: rabc.authorization.k8s.io/v1
kind: Role
metadata:
  namaspace: default
  name: configmap-update
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  resourceNames: ["my-configmap"]
  verbs: ["get","update"]
```

17.5 常见角色(role)授权的案例

17.5.1 允许读取核心 API 组的 Pod 资源

```
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get","list","watch"]
```

17.5.2 允许读写 apps API 组中的 deployment 资源

```
rules:
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get","list","watch","create","update","patch","delete"]
```

17.5.3 允许读取 Pod 以及读写 job 信息

```
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get","list","watch"]
- apiGroups: [""]
  resources: ["jobs"]
  verbs: ["get","list","watch","create","update","patch","delete"]
```

17.5.4 允许读取一个名为 my-config 的 ConfigMap（必须绑定到一个 RoleBinding 来限制到一个 Namespace 下的 ConfigMap）：

```
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  resourceNames: ["my-configmap"]
  verbs: ["get"]
```

17.5.5 读取核心组的 Node 资源（Node 属于集群级的资源，所以必须存在于 ClusterRole 中，并使用 ClusterRoleBinding 进行绑定）：

```
rules:
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get","list","watch"]
```

17.5.6 允许对非资源端点“/healthz”及其所有子路径进行 GET 和 POST 操作（必须使用 ClusterRole 和 ClusterRoleBinding）：

```
rules:
- nonResourceURLs: ["/healthz","/healthz/*"]
  verbs: ["get","post"]
```

17.6 常见的角色绑定示例

17.6.1 用户名 alice

```
subjects:
- kind: User
  name: alice
  apiGroup: rbac.authorization.k8s.io
```

17.6.2 组名 alice

```
subjects:
- kind: Group
  name: alice
  apiGroup: rbac.authorization.k8s.io
```

17.6.3 kube-system 命名空间中默认 Service Account

```
subjects:
- kind: ServiceAccount
  name: default
  namespace: kube-system
```

17.7 对 Service Account 的授权管理

Service Account 也是一种账号，是给运行在 Pod 里的进程提供了必要的身份证明。需要在 Pod 定义中指明引用的 Service Account，这样就可以对 Pod 的进行赋权操作。例如：pod 内可获取 rbac 命名空间的所有 Pod 资源，pod-reader-sc 的 Service Account 是绑定了名为 pod-read 的 Role

```
[root@master1 ~]# kubectl create ns rbac
[root@master1 ~]# kubectl create sa pod-reader-sc -n rbac
[root@master1 ~]# vim pod-rbac.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: rbac
spec:
  serviceAccountName: pod-reader-sc
  containers:
```

```
    - name: nginx
      image: nginx
      imagePullPolicy: IfNotPresent
      ports:
      - containerPort: 80
[root@master1 ~]# kubectl apply -f pod-rbac.yaml
```

（1）my-namespace 中的 my-sa Service Account 授予只读权限

```
kubectl create rolebinding my-sa-view --clusterrole=view --serviceaccount=my-namespace:my-sa --namespace=my-namespace
```

（2）为一个命名空间中名为 default 的 Service Account 授权

如果一个应用没有指定 serviceAccountName，则会使用名为 default 的 Service Account。注意，赋予 Service Account “default”的权限会让所有没有指定 serviceAccountName 的 Pod 都具有这些权限

例如，在 my-namespace 命名空间中为 Service Account “default”授予只读权限：

```
kubectl create rolebinding default-view --clusterrole=view --serviceaccount=my-namespace:default --namespace=my-namespace
```

（3）为命名空间中所有 Service Account 都授予一个角色

如果希望在一个命名空间中，任何 Service Account 应用都具有一个角色，则可以为这一命名空间的 Service Account 群组进行授权。

```
kubectl create rolebinding serviceaccounts-view --clusterrole=view --group=system:serviceaccounts:my-namespace --namespace=my-namespace
```

（4）为集群范围内所有 Service Account 都授予一个低权限角色

如果不想为每个命名空间管理授权，则可以把一个集群级别的角色赋给所有 Service Account。

```
kubectl create clusterrolebinding serviceaccounts-view --clusterrole=view --group=system:serviceaccounts
```

（5）为所有 Service Account 授予超级用户权限

```
kubectl create clusterrolebinding serviceaccounts-view --clusterrole=cluster-admin --group=system:serviceaccounts
```

17.8 使用 kubectl 命令行工具创建资源对象

（1）在命名空间 rbac 中为用户 es 授权 admin ClusterRole：

```
kubectl create rolebinding bob-admin-binding --clusterrole=admin --user=es --namespace=rbac
```

（2）在命名空间 rbac 中为名为 myapp 的 Service Account 授予 view ClusterRole：

```
kubctl create rolebinding myapp-role-binding --clusterrole=view --serviceaccount=rbac:myapp --namespace=rbac
```

（3）在全集群范围内为用户 root 授予 cluster-admin ClusterRole：

```
kubectl create clusterrolebinding cluster-binding --clusterrole=cluster-admin --user=root
```

（4）在全集群范围内为名为 myapp 的 Service Account 授予 view ClusterRole：

```
kubectl create clusterrolebinding service-account-binding --clusterrole=view --serviceaccount=myapp
```

yaml 文件进行 rbac 授权：https://kubernetes.io/zh/docs/reference/access-authn-authz/rbac/

17.9 限制不同的用户操作 k8s 集群

17.9.1 授权 kubectl 用户能查看 lucky-test 名称空间资源的权限

生成一个证书：

（1）生成一个私钥

cd /etc/kubernetes/pki/

umask 077; openssl genrsa -out lucky.key 2048

（2）生成一个证书请求

openssl req -new -key lucky.key -out lucky.csr -subj "/CN=lucky"

（3）生成一个证书

openssl x509 -req -in lucky.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out lucky.crt -days 3650

在 kubeconfig 下新增加一个 lucky 这个用户

（1）把 lucky 这个用户添加到 kubernetes 集群中，可以用来认证 apiserver 的连接

kubectl config set-credentials lucky --client-certificate=./lucky.crt --client-key=./lucky.key --embed-certs=true

（2）在 kubeconfig 下新增加一个 lucky 这个账号

kubectl config set-context lucky@kubernetes --cluster=kubernetes --user=lucky

（3）切换账号到 lucky，默认没有任何权限

kubectl config use-context lucky@kubernetes　　#切换后执行命令没有权限

kubectl config use-context kubernetes-admin@kubernetes　　#这个是集群用户，有任何权限

把 user 这个用户通过 rolebinding 绑定到 clusterrole 上，授予权限,权限只是在 lucky 这个名称空间有效

（1）把 lucky 这个用户通过 rolebinding 绑定到 clusterrole 上

kubectl create ns lucky-test

kubectl create rolebinding lucky -n lucky-test --clusterrole=cluster-admin --user=lucky

（2）切换到 lucky 这个用户

kubectl config use-context lucky@kubernetes

（3）测试是否有权限

kubectl get pods -n lucky-test

添加一个 lucky 的普通用户

useradd lucky

cp -ar /root/.kube /tmp/

修改/tmp/.kube/config 文件，把 kubernetes-admin 相关的删除，只留 lucky 用户

cp -ar /tmp/.kube/ /home/lucky/

chown -R lucky.lucky /home/lucky/

su - lucky

kubectl get pods -n lucky-test

退出 test 用户，需要在把集群环境切换成管理员权限

kubectl config use-context kubernetes-admin@kubernetes

17.9.2 授权 kubectl 用户能查看所有名称空间的 pod 的权限

生成一个证书：

（1）生成一个私钥

cd /etc/kubernetes/pki/

umask 077; openssl genrsa -out kubectl.key 2048

（2）生成一个证书请求

openssl req -new -key kubectl.key -out kubectl.csr -subj "/CN=kubectl"

（3）生成一个证书

openssl x509 -req -in kubectl.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out kubectl.crt -days 3650

在 kubeconfig 下新增加一个 kubectl 这个用户：

（1）把 lucky 这个用户添加到 kubernetes 集群中，可以用来认证 apiserver 的连接

kubectl config set-credentials kubectl --client-certificate=./kubectl.crt --client-key=./kubectl.key --embed-certs=true

（2）在 kubeconfig 下新增加一个 lucky 这个账号

kubectl config set-context kubectl@kubernetes --cluster=kubernetes --user=kubectl

(3)创建一个 clusterrole

```
# vim kubectl-clusterrole.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kubectl-get-pod
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
[root@master1 ~]# kubectl apply -f kubectl-clusterrole.yaml
```

（4）创建一个 clusterrolebinding

```
kubectl create clusterrolebinding kubectl-get-pods --clusterrole=kubectl-get-pod  --user=kubectl
```

添加一个 kubectl 的普通用户：

```
# useradd kubectl
# cp -ar /root/.kube /home/kubectl/
# vim /home/kubectl/.kube/config
```

把 kubernetes-admin 和 lucky 相关的删除，只留 kubectl 用户。

把 current-context 变成如下：current-context: kubectl@kubernetes

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data:
RTczc1BKLOxaT3RqOU5PR1U5tRmlXSzk1RjNTUTJVNOZGREJOSWgyZE1ZNnlFPQotCBDVJUSUZJQOFURSOtLSOtCg==
    server: https://192.168.40.180:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubectl
  name: kubectl@kubernetes
current-context: kubectl@kubernetes
kind: Config
preferences: {}
users:
- name: kubectl
  user:
    client-certificate-data:
AKak83NHNzeWxYZzltOVBTUVVmN3TcDl3NwpGQO9aUTZNakJBPTOKLSOtLS1FTkQgQOVSVElGSUNBVEUtLSOtLQo=
```

```
# chown -R kubectl.kubectl /home/kubectl/
# su - kubectl
$ kubectl get pods
$ kubectl get pods -n kube-system
```

17.10 准入控制

17.10.1 ResourceQuota 准入控制器

ResourceQuota 准入控制器是 k8s 上内置的准入控制器，默认该控制器是启用的状态，它主要作用是用来限制一个名称空间下的资源的使用，它能防止在一个名称空间下的 pod 被过多创建时，导致过多占用 k8s 资源，简单讲它是用来在名称空间级别限制用户的资源使用。

限制 cpu、内存、pod、deployment 数量：

```
[root@master1 ~]# kubectl create ns quota
[root@master1 ~]# vim resourcequota.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota-test
  namespace: quota
spec:
  hard:
    pods: "6"
    requests.cpu: "2"
    requests.memory: 2Gi
    limits.cpu: "4"
    limits.memory: 10Gi
    count/deployments.apps: "6"
    persistentvolumeclaims: "6"
[root@master1 ~]# kubectl apply -f resourcequota.yaml
```

资源清单 YAML 文件解读：

spec.hard 字段是用来定义对应名称空间下的资源限制规则；pods 用来限制在对应名称空间下的 pod 数量，requests.cpu 字段用来限制对应名称空间下所有 pod 的 cpu 资源的下限总和；requests.memory 用来限制对应名称空间下 pod 的内存资源的下限总和；limits.cpu 用来限制对应名称空间下的 pod.cpu 资源的上限总和，limits.memory 用来限制对应名称空间下 pod 内存资源上限总和；count/deployments.apps 用来限制对应名称空间下 apps 群组下的 deployments 的个数；

以上配置清单表示，在 quota 名称空间下运行的 pod 数量不能超过 6 个，所有 pod 的 cpu 资源下限总和不能大于 2 个核心，内存资源下限总和不能大于 2G，cpu 上限资源总和不能大于 4 个核心，内存上限总和不能超过 10G，apps 群组下的 deployments 控制器不能超过 6 个， pvc 个数不能超过 6 个；以上条件中任意一个条目不满足，都将无法在对应名称空间创建对应的资源。

```
[root@master1 ~]# vim quota-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: quota
  namespace: quota
spec:
```

```
  replicas: 7
  selector:
    matchLabels:
      app: quota
  template:
    metadata:
      labels:
         app: quota
    spec:
      containers:
      - name: myapp
        image: nginx
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
        resources:
         requests:
            cpu: 10m
            memory: 10Mi
         limits:
            cpu: 10m
            memory: 10Mi
[root@master1 ~]# kubectl apply -f quota-deployment.yaml
[root@master1 ~]# kubectl get pods -n quota
NAME                     READY   STATUS    RESTARTS   AGE
quota-6d5c459f69-4q86p   1/1     Running   0          8s
quota-6d5c459f69-7kchv   1/1     Running   0          8s
quota-6d5c459f69-dgzl7   1/1     Running   0          8s
quota-6d5c459f69-g6c8j   1/1     Running   0          9s
quota-6d5c459f69-hfdng   1/1     Running   0          9s
quota-6d5c459f69-nfb7p   1/1     Running   0          9s
```

限制存储空间大小：

```
[root@master1 ~]# vim resourcequota-2.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota-storage-test
  namespace: quota
spec:
  hard:
    requests.storage: "5Gi"
    persistentvolumeclaims: "5"
    requests.ephemeral-storage: "1Gi"
    limits.ephemeral-storage: "2Gi"
```

```
[root@master1 ~]# kubectl apply -f resourcequota-2.yaml
```

备注：requests.storage 用来限制对应名称空间下的存储下限总和，persistenvolumeclaims 用来限制 pvc 总数量，requests.ephemeral-storage 用来现在使用本地临时存储的下限总容量；limits.ephemeral-storage 用来限制使用本地临时存储上限总容量；以上配置表示在 default 名称空间下非停止状态的容器存储下限总容量不能超过 5G，pvc 的数量不能超过 5 个，本地临时存储下限容量不能超过 1G，上限不能超过 2G。

17.10.2 LimitRanger 准入控制器

LimitRanger 准入控制器是 k8s 上一个内置的准入控制器，LimitRange 是 k8s 上的一个标准资源，它主要用来定义在某个名称空间下限制 pod 或 pod 里的容器对 k8s 上的 cpu 和内存资源使用；它能够定义我们在某个名称空间下创建 pod 时使用的 cpu 和内存的上限和下限以及默认 cpu、内存的上下限。

如果我们创建 pod 时定义了资源上下限，但不满足 LimitRange 规则中定义的资源上下限，此时 LimitRanger 就会拒绝我们创建此 pod；如果我们在 LimitRange 规则中定义了默认的资源上下限制，我们创建资源没有指定其资源限制，它默认会使用 LimitRange 规则中的默认资源限制；同样的逻辑 LimitRanger 可以限制一个 pod 使用资源的上下限，它还可以限制 pod 中的容器的资源上下限，比限制 pod 更加精准；不管是针对 pod 还是 pod 里的容器，它始终只是限制单个 pod 资源使用。

```
[root@master1 ~]# vim limitrange.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: limit
---
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-memory
  namespace: limit
spec:
  limits:
  - default:
      cpu: 1000m
      memory: 1000Mi
    defaultRequest:
      cpu: 500m
      memory: 500Mi
    min:
      cpu: 500m
      memory: 500Mi
    max:
      cpu: 2000m
      memory: 2000Mi
    maxLimitRequestRatio:
      cpu: 4
      memory: 4
    type: Container
[root@master1 ~]# kubectl apply -f limitrange.yaml
```

备注：以上清单主要定义了两个资源，一个创建 limit 名称空间，一个是在对应 limit 名称空间下定义了 LimitRange 资

源；其中 LimitRange 资源的名称为 cpu-memory，default 字段用来指定默认容器资源上限值；defaultRequest 用来指定默认容器资源下限值；min 字段用来指定限制用户指定的资源下限不能小于对应资源的值；max 是用来限制用户指定资源上限值不能大于该值；maxLimitRequestRatio 字段用来指定资源的上限和下限的比值；即上限是下限的多少倍；type 是用来描述对应资源限制的级别，该字段有两个值 pod 和 container。

上述资源清单表示在该名称空间下创建 pod 时，默认不指定其容器的资源限制，就限制对应容器最少要有 0.5 个核心的 cpu 和 500M 的内存；最大为 1 个核心 cpu,1g 内存；如果我们手动定义了容器的资源限制，那么对应资源限制最小不能小于 cpu 为 0.5 个核心，内存为 500M，最大不能超过 cpu 为 2 个核心，内存为 2000M；

如果我们在创建 pod 时，只指定了容器的资源上限或下限，那么上限最大是下限的的 4 倍，如果指定 cpu 上限为 2000m 那么下限一定不会小于 500m，如果只指定了 cpu 下限为 500m 那么上限最大不会超过 2000m，对于内存也是同样的逻辑。

limitrange 使用案例：

（1）在 limit 名称空间创建 pod，不指定资源，看看是否会被 limitrange 规则自动附加其资源限制？

```
[root@master1 ~]# vim pod-limit.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-demo
  namespace: limit
spec:
  containers:
  - image: nginx
    imagePullPolicy: IfNotPresent
    name: nginx
[root@master1 ~]# kubectl apply -f pod-limit.yaml
[root@master1 ~]# kubectl describe pods nginx-pod-demo -n limit
```

显示如下：

```
    Limits:
      cpu:     1
      memory:  1000Mi
    Requests:
      cpu:        500m
      memory:     500Mi
```

通过上面结果可以看到我们在 limit 名称空间下创建的 pod 没有指定其容器资源限制，创建 pod 后，其内部容器自动就有了默认的资源限制；其大小就是我们在定义 LimitRange 规则中的 default 和 defaultRequest 字段中指定的资源限制。

（2）创建 pod，指定 cpu 请求是 100m，看看是否允许创建

```
[root@master1 ~]# vim pod-request.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-request
  namespace: limit
spec:
  containers:
```

```
    - image: nginx
      imagePullPolicy: IfNotPresent
      name: nginx
      resources:
        requests:
          cpu: 100m
[root@master1 ~]# kubectl apply -f pod-request.yaml
Error from server (Forbidden): error when creating "pod-request.yaml": pods "pod-request" is forbidden: [minimum cpu usage per Container is 500m, but request is 100m, cpu max limit to request ratio per Container is 4, but provided ratio is 10.000000]
```

18 Ingress 和 Ingress Controller 概述

18.1 Ingress 介绍

四层代理：Service 可以通过标签选择器找到它所关联的 Pod。但是属于四层代理，只能基于 IP 和端口代理。

Ingress 官网定义：Ingress 可以把进入到集群内部的请求转发到集群中的一些服务上，从而可以把服务映射到集群外部。Ingress 能把集群内 Service 配置成外网能够访问的 URL，流量负载均衡，提供基于域名访问的虚拟主机等。

Ingress 简单的理解就是你原来需要改 Nginx 配置，然后配置各种域名对应哪个 Service，现在把这个动作抽象出来，变成一个 Ingress 对象,你可以用 yaml 创建,每次不要去改 Nginx 了,直接改 yaml 然后创建/更新就行了;那么问题来了:”Nginx 该怎么处理？”

Ingress Controller 就是解决“Nginx 的处理方式”的；Ingress Controller 通过与 Kubernetes API 交互，动态感知集群中 Ingress 规则变化，然后读取他，按照他自己模板生成一段 Nginx 配置，再写到 Nginx Pod 里，最后 reload 一下。

18.2 Ingress Controller 介绍

Ingress Controller 是一个七层负载均衡调度器，客户端的请求先到达这个七层负载均衡调度器，由七层负载均衡器在反向代理到后端 pod，常见的七层负载均衡器有 nginx、traefik，以我们熟悉的 nginx 为例，假如请求到达 nginx，会通过 upstream 反向代理到后端 pod 应用，但是后端 pod 的 ip 地址是一直在变化的，因此在后端 pod 前需要加一个 service，这个 service 只是起到分组的作用，那么我们 upstream 只需要填写 service 地址即可

18.3 Ingress 和 Ingress Controller 总结

Ingress Controller 可以理解为控制器，它通过不断的跟 Kubernetes API 交互，实时获取后端 Service、Pod 的变化，比如新增、删除等，结合 Ingress 定义的规则生成配置，然后动态更新上边的 Nginx 或者 trafik 负载均衡器，并刷新使配置生效，来达到服务自动发现的作用。

Ingress 则是定义规则,通过它定义某个域名的请求过来之后转发到集群中指定的 Service。它可以通过 Yaml 文件定义，可以给一个或多个 Service 定义一个或多个 Ingress 规则。

18.4 使用 Ingress Controller 代理 k8s 内部应用的流程

（1）部署 Ingress controller，我们 ingress controller 使用的是 nginx

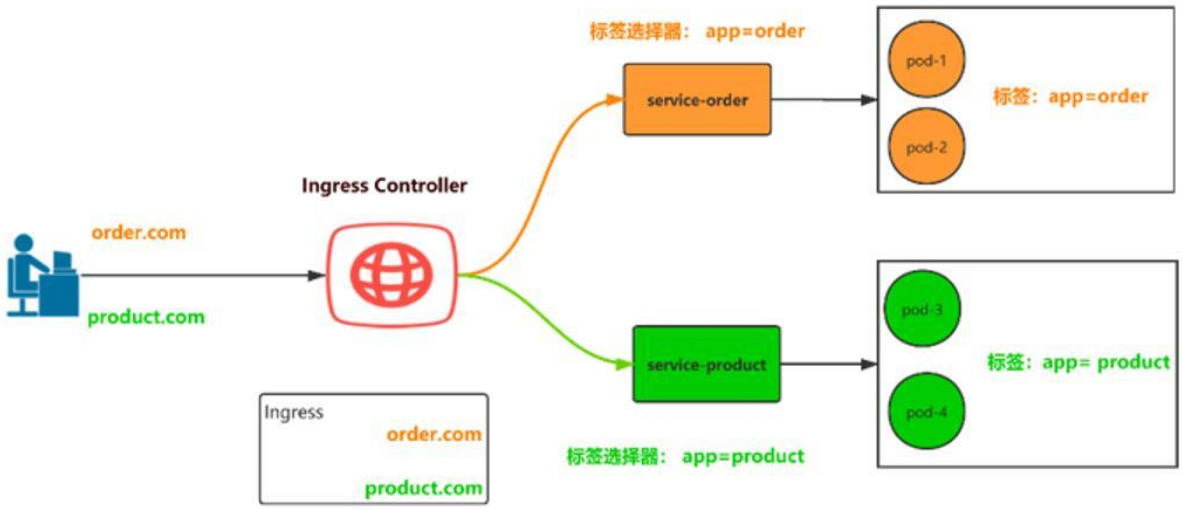
（2）创建 Pod 应用，可以通过控制器创建 pod

（3）创建 Service，用来分组 pod

（4）创建 Ingress http，测试通过 http 访问应用

（5）创建 Ingress https，测试通过 https 访问应用

使用七层负载均衡调度器 ingress controller 时，当客户端访问 k8s 集群内部的应用时，数据包走向如图流程所示：



18.5 Ingress-controller 高可用

Ingress Controller 是集群流量的接入层，对它做高可用非常重要，可以基于 keepalive 实现 nginx-ingress-controller 高可用，具体实现如下：

Ingress-controller 根据 Deployment + nodeSeletor + pod 反亲和性方式部署在 k8s 指定的两个 work 节点，nginx-ingress-controller 这个 pod 共享宿主机 ip，然后通过 keepalive+lvs 实现 nginx-ingress-controller 高可用

参考：https://github.com/kubernetes/ingress-nginx

https://github.com/kubernetes/ingress-nginx/tree/main/deploy/static/provider/baremetal

```
[root@master1 ~]# kubectl label node node1 kubernetes.io/ingress=nginx
[root@master1 ~]# kubectl label node node2 kubernetes.io/ingress=nginx
[root@node1 ~]# ctr -n=k8s.io images import ingress-nginx-controllerv1.1.0.tar.gz
[root@node1 ~]# ctr -n=k8s.io images import kube-webhook-certgen-v1.1.0.tar.gz
[root@node2 ~]# ctr -n=k8s.io images import ingress-nginx-controllerv1.1.0.tar.gz
[root@node2 ~]# ctr -n=k8s.io images import kube-webhook-certgen-v1.1.0.tar.gz

[root@master1 ~]# kubectl apply -f ingress-deploy.yaml
[root@master1 ~]# kubectl get pods -n ingress-nginx -o wide
NAME                                   READY   STATUS      RESTARTS   AGE    IP              NODE
ingress-nginx-admission-create-k4fmn   0/1     Completed   0          103s   10.244.1.12     node1
ingress-nginx-admission-patch-x87h8    0/1     Completed   1          103s   10.244.1.11     node1
ingress-nginx-controller-6c8ffbbfcf-cjj9t   1/1     Running     0          103s   192.168.40.181   node1
ingress-nginx-controller-6c8ffbbfcf-wpt26   1/1     Running     0          103s   192.168.40.182   node2
```

18.5.1 通过 keepalive+nginx 实现 nginx-ingress-controller 高可用

1、安装 nginx 主备：在 node1 和 node2 上做 nginx 主备安装

```
# yum install epel-release nginx nginx-mod-stream keepalived -y
```

2、修改 nginx 配置文件(主备一样)，node1 和 node2 均操作

```
# vim /etc/nginx/nginx.conf
user nginx;
worker_processes auto;
error_log /var/log/nginx/error.log;
```

```
pid /run/nginx.pid;
include /usr/share/nginx/modules/*.conf;
events {
    worker_connections 1024;
}
# 四层负载均衡，为两台 Master apiserver 组件提供负载均衡
stream {
    log_format  main  '$remote_addr $upstream_addr - [$time_local] $status $upstream_bytes_sent';
    access_log  /var/log/nginx/k8s-access.log  main;
    upstream k8s-ingress {
       server 192.168.40.181:80;   # Master1 APISERVER IP:PORT
       server 192.168.40.182:80;   # Master2 APISERVER IP:PORT
    }
    server {
       listen 30080;
       proxy_pass k8s-ingress;
    }
}
http {
    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';
    access_log  /var/log/nginx/access.log  main;
    sendfile            on;
    tcp_nopush          on;
    tcp_nodelay         on;
    keepalive_timeout   65;
    types_hash_max_size 2048;
    include             /etc/nginx/mime.types;
    default_type        application/octet-stream;
}
```

注意：nginx 监听端口变成大于 30000 的端口，比方说 30080,这样访问域名:30080 就可以了，必须是满足大于 30000 以上，才能代理 ingress-controller。

3、keepalive 配置

主 keepalived：

```
[root@node1 ~]# vim /etc/keepalived/keepalived.conf
global_defs {
   notification_email {
     acassen@firewall.loc
     failover@firewall.loc
     sysadmin@firewall.loc
   }
   notification_email_from Alexandre.Cassen@firewall.loc
   smtp_server 127.0.0.1
```

```
   smtp_connect_timeout 30
   router_id NGINX_MASTER
}
vrrp_script check_nginx {
    script "/etc/keepalived/check_nginx.sh"
}
vrrp_instance VI_1 {
    state MASTER
    interface eth0          #修改为实际网卡名
    virtual_router_id 51    #VRRP 路由 ID 实例，每个实例是唯一的
    priority 100            #优先级，备服务器设置 90
    advert_int 1            #指定 VRRP 心跳包通告间隔时间，默认 1 秒
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    # 虚拟 IP
    virtual_ipaddress {
        192.168.40.199
    }
    track_script {
        check_nginx
    }
}
[root@node1 ~]# vim /etc/keepalived/check_nginx.sh   //编辑 keepalived 存活检测脚本
#!/bin/bash
counter=`ps -C nginx --no-header | wc -l`
if [ $counter -eq 0 ]; then
    service nginx start
    sleep 2
    counter=`ps -C nginx --no-header | wc -l`
    if [ $counter -eq 0 ]; then
        service  keepalived stop
    fi
fi
[root@node1 ~]# chmod +x /etc/keepalived/check_nginx.sh
```

备 keepalive

```
[root@node2 ~]# vim /etc/keepalived/keepalived.conf
global_defs {
   notification_email {
     acassen@firewall.loc
     failover@firewall.loc
     sysadmin@firewall.loc
```

```
   }
   notification_email_from Alexandre.Cassen@firewall.loc
   smtp_server 127.0.0.1
   smtp_connect_timeout 30
   router_id NGINX_BACKUP
}
vrrp_script check_nginx {
    script "/etc/keepalived/check_nginx.sh"
}
vrrp_instance VI_1 {
    state BACKUP
    interface eth0  # 修改为实际网卡名
    virtual_router_id 51 # VRRP 路由 ID 实例，每个实例是唯一的
    priority 90     # 优先级，备服务器设置 90
    advert_int 1    # 指定 VRRP 心跳包通告间隔时间，默认 1 秒
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    # 虚拟 IP
    virtual_ipaddress {
        192.168.40.199
    }
    track_script {
        check_nginx
    }
}
[root@node2 ~]# vim /etc/keepalived/check_nginx.sh
#!/bin/bash
counter=`ps -C nginx --no-header | wc -l`
if [ $counter -eq 0 ]; then
    service nginx start
    sleep 2
    counter=`ps -C nginx --no-header | wc -l`
    if [ $counter -eq 0 ]; then
        service  keepalived stop
    fi
fi
[root@node2 ~]# chmod +x /etc/keepalived/check_nginx.sh
#注：keepalived 根据脚本返回状态码（0 为工作正常，非 0 不正常）判断是否故障转移。
4、启动服务：node1+node2
# systemctl daemon-reload
# systemctl enable nginx keepalived
# systemctl start nginx keepalived
```

5、测试 vip 是否绑定成功

```
# ip addr
......
    inet 192.168.1.199/24 scope global secondary ens33
......
```

6、测试 keepalived

停掉 node1 上的 keepalived，Vip 会漂移到 node2：

```
[root@node1 ~]# service keepalived stop
[root@node2 ~]# ip addr
......
    inet 192.168.1.199/24 scope global secondary ens33
```

启动 node1 上的 keepalived，Vip 又会漂移到 master1：

```
[root@node1 ~]# service keepalived start
[root@node1 ~]# ip addr
......
    inet 192.168.1.199/24 scope global secondary ens33
```

18.5.2 测试 Ingress HTTP 代理 k8s 内部站点

1.部署后端 tomcat 服务

```
[root@master1 ~]# vim ingress-demo.yaml
apiVersion: v1
kind: Service
metadata:
  name: tomcat
  namespace: default
spec:
  selector:
    app: tomcat
    release: canary
  ports:
  - name: http
    targetPort: 8080
    port: 8080
  - name: ajp
    targetPort: 8009
    port: 8009
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tomcat-deploy
  namespace: default
spec:
  replicas: 2
  selector:
```

```
    matchLabels:
      app: tomcat
      release: canary
  template:
    metadata:
      labels:
        app: tomcat
        release: canary
    spec:
      containers:
      - name: tomcat
        image: tomcat:8.5-jre8-alpine
        imagePullPolicy: IfNotPresent
        ports:
        - name: http
          containerPort: 8080
          name: ajp
          containerPort: 8009
[root@master1 ~]# kubectl apply -f ingress-demo.yaml      #更新资源清单 yaml 文件
[root@master1 ~]# kubectl get pods -l app=tomcat      #查看 pod 是否部署成功
NAME                              READY   STATUS    RESTARTS   AGE
tomcat-deploy-66b67fcf7b-9h9qp    1/1     Running   0          32s
tomcat-deploy-66b67fcf7b-hxtkm    1/1     Running   0          32s
```

2、编写 ingress 规则

```
[root@master1 ~]# vim ingress-myapp.yaml      #编写 ingress 的配置清单
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-myapp
  namespace: default
#  annotations:    #1.23 用这种注解的模式
#    kubernetes.io/ingress.class: "nginx"
spec:
  ingressClassName: nginx      #1.25 用 ingressClassName 的模式
  rules:
  - host: tomcat.lucky.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: tomcat
            port:
```

```
              number: 8080
[root@master1 ~]# kubectl apply -f ingress-myapp.yaml     #更新 yaml 文件：
```

若出现如下报错：

```
[root@xianchaomaster1 ingress]# kubectl apply -f ingress-myapp.yaml
Error from server (InternalError): error when creating "ingress-myapp.yaml": Internal error occurred: failed calling webhook "validat
e.nginx.ingress.kubernetes.io": failed to call webhook: Post "https://ingress-nginx-controller-admission.ingress-nginx.svc:443/networ
king/v1/ingresses?timeout=10s": context deadline exceeded
```

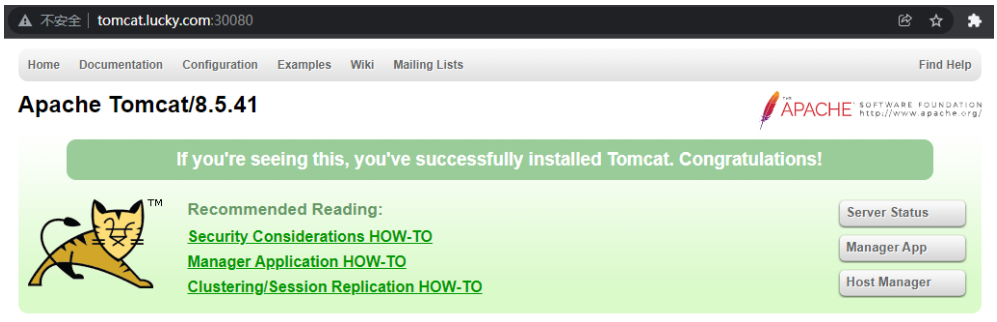
通过如下命令解决：

```
[root@master1 ~]# kubectl delete -A ValidatingWebhookConfiguration ingress-nginx-admission
[root@master1 ~]# kubectl apply -f ingress-myapp.yaml
[root@master1 ~]# kubectl describe ingress ingress-myapp     #查看 ingress-myapp 的详细信息
Name:             ingress-myapp
Namespace:        default
Address:
Default backend:  default-http-backend:80 (10.244.187.118:8080)
Rules:
  Host              Path  Backends
  ----              ----  --------
  tomcat.lucky.com
                    tomcat:8080 (10.244.209.172:8080,10.244.209.173:8080)
Annotations:        kubernetes.io/ingress.class: nginx
Events:
  Type    Reason  Age   From                      Message
  ----    ------  ----  ----                      -------
  Normal  CREATE  22s   nginx-ingress-controller  Ingress default/ingress-myapp
```

修改电脑本地的 host 文件，增加如下一行：

```
192.168.40.199   tomcat.lucky.com
```

浏览器访问 tomcat.lucky.com，出现如下页面(自己测试只有域名能访问，用 ip+端口访问不通)：



总结：

通过 deployment+nodeSelector+pod 反亲和性实现 ingress-controller 在 node1 和 node2 调度。

Keeplaive+nginx 实现 ingress-controller 高可用。

测试 ingress 七层代理是否正常。

18.5.3 测试 Ingress HTTPS 代理 k8s 内部站点

1、准备证书

```
[root@master1 ~]# cd /root
[root@master1 ~]# openssl genrsa -out tls.key 2048
[root@master1 ~]# openssl req -new -x509 -key tls.key -out tls.crt -subj
/C=CN/ST=Beijing/L=Beijing/O=DevOps/CN=tomcat8.lucky.com
```

2、生成 secret

```
[root@master1 ~]# kubectl create secret tls tomcat-ingress-secret --cert=tls.crt --key=tls.key
[root@master1 ~]# kubectl get secret
tomcat-ingress-secret   kubernetes.io/tls   2      8s
```

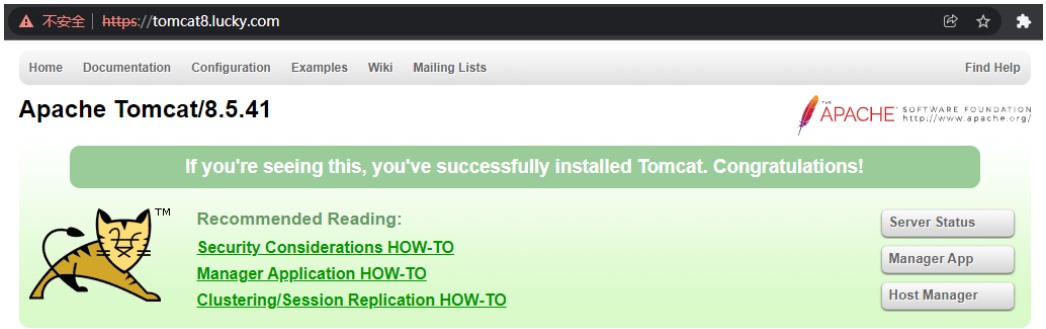
3、创建 Ingress

```
[root@master1 ~]# vim ingress-tomcat-tls.yaml      #编写 ingress 的配置清单
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-tomcat-tls
  namespace: default
spec:
  ingressClassName: nginx      #1.25 用 ingressClassName 的模式
  tls:
  - hosts:
    - tomcat8.lucky.com
    secretName: tomcat-ingress-secret
  rules:
  - host: tomcat8.lucky.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: tomcat
            port:
              number: 8080
[root@master1 ~]# kubectl apply -f ingress-tomcat-tls.yaml
```

修改电脑本地的 host 文件，增加如下一行：

192.168.40.199 tomcat8.lucky.com

浏览器访问 tomcat8.lucky.com，出现如下页面：



18.5.4 同一个 k8s 搭建多套 Ingress-controller

ingress 可以简单理解为 service 的 service，他通过独立的 ingress 对象来制定请求转发的规则，把请求路由到一个或多个 service 中。这样就把服务与请求规则解耦了，可以从业务维度统一考虑业务的暴露，而不用为每个 service 单独考虑。

在同一个 k8s 集群里，部署两个 ingress nginx。一个 deploy 部署给 A 的 API 网关项目用。另一个 daemonset 部署给其它项目作域名访问用。这两个项目的更新频率和用法不一致，暂时不用合成一个。

为了满足多租户场景，需要在 k8s 集群部署多个 ingress-controller，给不同用户不同环境使用。

主要参数设置：

```
containers:
- name: nginx-ingress-controller
  image: registry.cn-hangzhou.aliyuncs.com/google_containers/nginx-ingress-controller:v1.1.0
  args:
  - /nginx-ingress-controller
  - --ingress-class=ngx-ds
```

注意：--ingress-class 设置该 Ingress Controller 可监听的目标 Ingress Class 标识；注意：同一个集群中不同套 Ingress Controller 监听的 Ingress Class 标识必须唯一，且不能设置为 nginx 关键字（其是集群默认 Ingress Controller 的监听标识）；

创建 Ingress 规则：

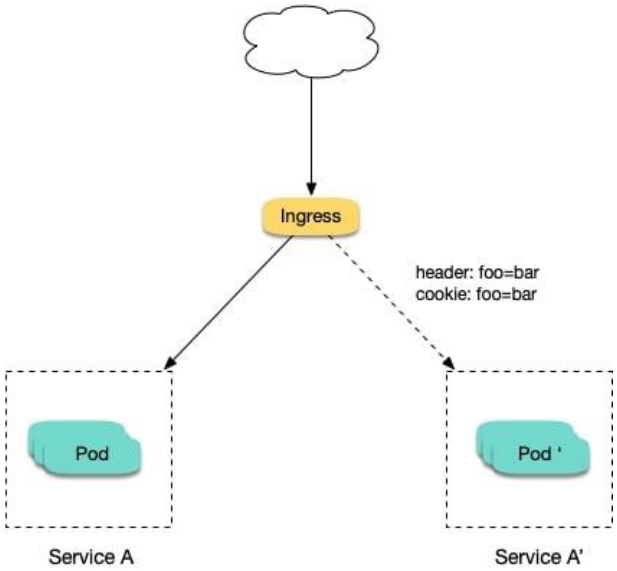
```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-myapp
  namespace: default
  annotations:
    kubernetes.io/ingress.class: "ngx-ds"
spec:
  rules:
  - host: tomcat.lucky.com
    http:
      paths:
      - path: /
        pathType:  Prefix
        backend:
          service:
            name: tomcat
            port:
              number: 8080
```

18.6 通过 Ingress-nginx 实现灰度发布

场景一：将新版本灰度给部分用户

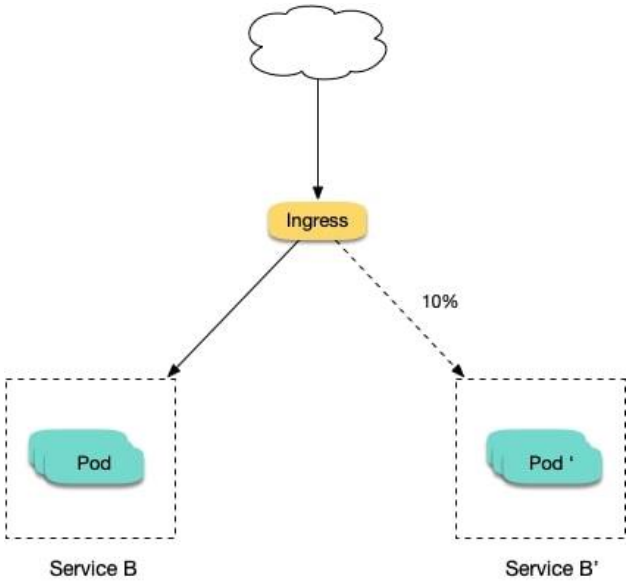
假设线上运行了一套对外提供 7 层服务的 Service A 服务，后来开发了个新版本 Service A’想要上线，但又不想直接替换掉原来的 Service A，希望先灰度一小部分用户，等运行一段时间足够稳定了再逐渐全量上线新版本，最后平滑下线旧版本。这个时候就可以利用 Nginx Ingress 基于 Header 或 Cookie 进行流量切分的策略来发布，业务使用 Header 或 Cookie 来标识不同类型的用户，我们通过配置 Ingress 来实现让带有指定 Header 或 Cookie 的请求被转发到新版本，其它的仍然转

发到旧版本，从而实现将新版本灰度给部分用户。



场景二：切一定比例的流量给新版本

假设线上运行了一套对外提供7层服务的Service B服务，后来修复了一些问题，需要灰度上线一个新版本Service B'，但又不想直接替换掉原来的Service B，而是让先切10%的流量到新版本，等观察一段时间稳定后再逐渐加大新版本的流量比例直至完全替换旧版本，最后再滑下线旧版本，从而实现切一定比例的流量给新版本。



Ingress-Nginx是一个K8S ingress工具，支持配置Ingress Annotations来实现不同场景下的灰度发布和测试。 Nginx Annotations支持以下几种Canary规则：

假设我们现在部署了两个版本的服务，老版本和canary版本。

nginx.ingress.kubernetes.io/canary-by-header：基于Request Header的流量切分，适用于灰度发布以及 A/B 测试。当Request Header设置为always时，请求将会被一直发送到Canary版本；当Request Header设置为never时，请求不会被发送到Canary入口。

nginx.ingress.kubernetes.io/canary-by-header-value：要匹配的Request Header的值，用于通知Ingress将请求路由到Canary Ingress中指定的服务。当Request Header设置为此值时，它将被路由到Canary入口。

nginx.ingress.kubernetes.io/canary-weight：基于服务权重的流量切分，适用于蓝绿部署，权重范围0 - 100按百分比将请求路由到Canary Ingress中指定的服务。权重为0意味着该金丝雀规则不会向Canary入口的服务发送任何请求。权

重为 60 意味着 60%流量转到 canary。权重为 100 意味着所有请求都将被发送到 Canary 入口。

nginx.ingress.kubernetes.io/canary-by-cookie：基于 Cookie 的流量切分，适用于灰度发布与 A/B 测试。用于通知 Ingress 将请求路由到 Canary Ingress 中指定的服务的 cookie。当 cookie 值设置为 always 时，它将被路由到 Canary 入口；当 cookie 值设置为 never 时，请求不会被发送到 Canary 入口。

部署两个版本的服务，这里以简单的 nginx 为例，先部署一个 v1 版本:

```
[root@node1 ~]# ctr -n=k8s.io images import openresty.tar.gz
[root@node2 ~]# ctr -n=k8s.io images import openresty.tar.gz
[root@master1 ~]# vim v1.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
      version: v1
  template:
    metadata:
      labels:
        app: nginx
        version: v1
    spec:
      containers:
      - name: nginx
        image: "openresty/openresty:centos"
        imagePullPolicy: IfNotPresent
        ports:
        - name: http
          protocol: TCP
          containerPort: 80
        volumeMounts:
        - mountPath: /usr/local/openresty/nginx/conf/nginx.conf
          name: config
          subPath: nginx.conf
      volumes:
      - name: config
        configMap:
          name: nginx-v1
---
apiVersion: v1
kind: ConfigMap
```

```
metadata:
  labels:
    app: nginx
    version: v1
  name: nginx-v1
data:
  nginx.conf: |
    worker_processes  1;
    events {
        accept_mutex on;
        multi_accept on;
        use epoll;
        worker_connections  1024;
    }
    http {
        ignore_invalid_headers off;
        server {
            listen 80;
            location / {
                access_by_lua '
                    local header_str = ngx.say("nginx-v1")
                ';
            }
        }
    }
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-v1
spec:
  type: ClusterIP
  ports:
  - port: 80
    protocol: TCP
    name: http
  selector:
    app: nginx
    version: v1
[root@master1~]# kubectl apply -f v1.yaml
```

再部署一个 v2 版本:

```
[root@master1 ~]# vim v2.yaml
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: nginx-v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
      version: v2
  template:
    metadata:
      labels:
        app: nginx
        version: v2
    spec:
      containers:
      - name: nginx
        image: "openresty/openresty:centos"
        imagePullPolicy: IfNotPresent
        ports:
        - name: http
          protocol: TCP
          containerPort: 80
        volumeMounts:
        - mountPath: /usr/local/openresty/nginx/conf/nginx.conf
          name: config
          subPath: nginx.conf
      volumes:
      - name: config
        configMap:
          name: nginx-v2
---
apiVersion: v1
kind: ConfigMap
metadata:
  labels:
    app: nginx
    version: v2
  name: nginx-v2
data:
  nginx.conf: |
    worker_processes  1;
    events {
        accept_mutex on;
```

```
        multi_accept on;
        use epoll;
        worker_connections  1024;
    }
    http {
        ignore_invalid_headers off;
        server {
            listen 80;
            location / {
                access_by_lua '
                    local header_str = ngx.say("nginx-v2")
                ';
            }
        }
    }
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-v2
spec:
  type: ClusterIP
  ports:
  - port: 80
    protocol: TCP
    name: http
  selector:
    app: nginx
    version: v2
[root@master1 ~]# kubectl apply -f v2.yaml
```

再创建一个 Ingress，对外暴露服务，指向 v1 版本的服务:

```
[root@master1 ~]# vim v1-ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx
spec:
  ingressClassName: nginx
  rules:
  - host: canary.example.com
    http:
      paths:
      - path: /  #配置访问路径，如果通过 url 进行转发，需要修改；空默认为访问的路径为"/"
```

```
        pathType: Prefix
        backend:  #配置后端服务
         service:
           name: nginx-v1
           port:
            number: 80
[root@master1 ~]# kubectl apply -f v1-ingress.yaml
[root@master1 ~]# curl -H "Host: canary.example.com" http://192.168.40.199
nginx-v1
```

基于 Header 的流量切分：

创建 Canary Ingress，指定 v2 版本的后端服务，且加上一些 annotation，实现仅将带有名为 Region 且值为 cd 或 sz 的请求头的请求转发给当前 Canary Ingress，模拟灰度新版本给成都和深圳地域的用户：

```
[root@master1 ~]# vim v2-ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/canary: "true"
    nginx.ingress.kubernetes.io/canary-by-header: "Region"
    nginx.ingress.kubernetes.io/canary-by-header-pattern: "cd|sz"
  name: nginx-canary
spec:
  ingressClassName: nginx
  rules:
  - host: canary.example.com
    http:
      paths:
      - path: /  #配置访问路径，如果通过 url 进行转发，需要修改；空默认为访问的路径为"/"
        pathType: Prefix
        backend:  #配置后端服务
         service:
           name: nginx-v2
           port:
            number: 80
[root@master1 ~]# kubectl apply -f v2-ingress.yaml
[root@master1 ~]# curl -H "Host: canary.example.com" -H "Region: cd" http://192.168.40.199
nginx-v2
[root@master1 ~]# curl -H "Host: canary.example.com" -H "Region: bj" http://192.168.40.199
nginx-v1
```

可以看到，只有 header Region 为 cd 或 sz 的请求才由 v2 版本服务响应。

基于 Cookie 的流量切分：

与前面 Header 类似，不过使用 Cookie 就无法自定义 value 了，这里以模拟灰度成都地域用户为例，仅将带有名为

user_from_cd 的 cookie 的请求转发给当前 Canary Ingress。先删除前面基于 Header 的流量切分的 Canary Ingress，然后创建下面新的 Canary Ingress:

```
[root@master1 ~]# kubectl delete -f v2-ingress.yaml
[root@master1 ~]# vim v2-cookie.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/canary: "true"
    nginx.ingress.kubernetes.io/canary-by-cookie: "user_from_cd"
  name: nginx-canary
spec:
  ingressClassName: nginx
  rules:
  - host: canary.example.com
    http:
      paths:
      - path: /  #配置访问路径，如果通过 url 进行转发，需要修改；空默认为访问的路径为"/"
        pathType:  Prefix
        backend:  #配置后端服务
         service:
           name: nginx-v2
           port:
            number: 80
[root@master1 ~]# kubectl apply -f v1-cookie.yaml
[root@master1 ~]# curl -s -H "Host: canary.example.com" --cookie "user_from_cd=always"
http://192.168.40.199
nginx-v2
[root@master1 ~]# curl -s -H "Host: canary.example.com" --cookie "user_from_bj=always"
http://192.168.40.199
nginx-v1
[root@master1 ~]# curl -s -H "Host: canary.example.com" http://192.168.40.199
nginx-v1
```

可以看到，只有 cookie user_from_cd 为 always 的请求才由 v2 版本的服务响应。

基于服务权重的流量切分:

基于服务权重的 Canary Ingress 就简单了，直接定义需要导入的流量比例，这里以导入 10% 流量到 v2 版本为例（如果有，先删除之前的 Canary Ingress):

```
[root@master1 ~]# kubectl delete -f v2-cookie.yaml
[root@master1 ~]# vim v2-weight.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
```

```
    nginx.ingress.kubernetes.io/canary: "true"
    nginx.ingress.kubernetes.io/canary-weight: "10"
  name: nginx-canary
spec:
  ingressClassName: nginx
  rules:
  - host: canary.example.com
    http:
      paths:
      - path: /  #配置访问路径，如果通过 url 进行转发，需要修改；空默认为访问的路径为"/"
        pathType:  Prefix
        backend:  #配置后端服务
         service:
           name: nginx-v2
           port:
            number: 80
[root@master1 ~]# kubectl apply -f v2-weight.yaml
[root@master1 ~]# for i in {1..10}; do curl -H "Host: canary.example.com" http://192.168.40.181; done;
```

返回如下结果：

```
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v2
nginx-v1
nginx-v1
nginx-v1
```

可以看到，大概只有十分之一的几率由 v2 版本的服务响应，符合 10% 服务权重的设置