主讲老师：Fox

> 有道笔记地址：https://note.youdao.com/s/dmuOjOUO

1. K8S 概览

1.1 K8S 是什么

K8S官网文档：**https://kubernetes.io/zh/docs/home/**

K8S 是Kubernetes的全称，源于希腊语，意为“舵手”或“飞行员”。Kubernetes 是用于自动部署、扩缩和管理容器化应用程序的开源系统。 Kubernetes 源自Google 15 年生产环境的运维经验，同时凝聚了社区的最佳创意和实践。

> **Docker**：作为开源的应用容器引擎，可以把应用程序和其相关依赖打包生成一个 Image 镜像文件，是一个标准的运行环境，提供可持续交付的能力；
> **Kubernetes**：作为开源的容器编排引擎，用来对容器化应用进行自动化部署、 扩缩和管理；

1.2 K8S核心特性

- 服务发现与负载均衡：无需修改你的应用程序即可使用陌生的服务发现机制。
- 存储编排：自动挂载所选存储系统，包括本地存储。
- Secret和配置管理：部署更新Secrets和应用程序的配置时不必重新构建容器镜像，且不必将软件堆栈配置中的秘密信息暴露出来。
- 批量执行：除了服务之外，Kubernetes还可以管理你的批处理和CI工作负载，在期望时替换掉失效的容器。
- 水平扩缩：使用一个简单的命令、一个UI或基于CPU使用情况自动对应用程序进行扩缩。
- 自动化上线和回滚：Kubernetes会分步骤地将针对应用或其配置的更改上线，同时监视应用程序运行状况以确保你不会同时终止所有实例。
- 自动装箱：根据资源需求和其他约束自动放置容器，同时避免影响可用性。
- 自我修复：重新启动失败的容器，在节点死亡时替换并重新调度容器，杀死不响应用户定义的健康检查的容器。

1.3 K8S 核心架构

我们已经知道了 K8S 的核心功能：自动化运维管理多个容器化程序。那么 K8S 怎么做到的呢？这里，我们从宏观架构上来学习 K8S 的设计思想。首先看下图：

K8S 是属于**Master-Worker架构**，即有 Master 节点负责核心的调度、管理和运维，Worker 节点则执行用户的程序。但是在 K8S 中，主节点一般被称为**Master Node** ，而从节点则被称为**Worker Node 或者 Node**。

注意：Master Node 和 Worker Node 是分别安装了 K8S 的 Master 和 Woker 组件的实体服务器，每个 Node 都对应了一台实体服务器（虽然 Master Node 可以和其中一个 Worker Node 安装在同

一台服务器，但是建议 Master Node 单独部署），**所有 Master Node 和 Worker Node 组成了 K8S 集群**，同一个集群可能存在多个 Master Node 和 Worker Node。

首先来看**Master Node**都有哪些组件：

- **kube-apiserver。K8S 的请求入口服务。**API Server 负责接收 K8S 所有请求（来自 UI 界面或者 CLI 命令行工具），然后，API Server 根据用户的具体请求，去通知其他组件干活。
- **Scheduler。K8S 所有 Worker Node 的调度器。**当用户要部署服务时，Scheduler 会选择最合适的 Worker Node（服务器）来部署。
- **Controller Manager。K8S 所有 Worker Node 的监控器。**Controller Manager 有很多具体的 Controller，Node Controller、Service Controller、Volume Controller 等。Controller 负责监控和调整在 Worker Node 上部署的服务的状态，比如用户要求 A 服务部署 2 个副本，那么当其中一个服务挂了的时候，Controller 会马上调整，让 Scheduler 再选择一个 Worker Node 重新部署服务。
- **etcd。K8S 的存储服务。**etcd 存储了 K8S 的关键配置和用户配置，K8S 中仅 API Server 才具备读写权限，其他组件必须通过 API Server 的接口才能读写数据。

接着来看**Worker Node**的组件：

- **Kubelet。Worker Node 的监视器，以及与 Master Node 的通讯器。**Kubelet 是 Master Node 安插在 Worker Node 上的“眼线”，它会定期向 Master Node 汇报自己 Node 上运行的服务的状态，并接受来自 Master Node 的指示采取调整措施。负责控制所有容器的启动停止，保证节点工作正常。
- **Kube-Proxy。K8S 的网络代理。**Kube-Proxy 负责 Node 在 K8S 的网络通讯、以及对外部网络流量的负载均衡。
- **Container Runtime。Worker Node 的运行环境。**即安装了容器化所需的软件环境确保容器化程序能够跑起来，比如 Docker Engine运行环境。

1.4 K8S集群安装

官方文档： https://kubernetes.io/zh-cn/docs/setup/production-environment/tools/kubeadm/install-kubeadm/
k8s v1.20.1 https://note.youdao.com/s/MignGqZj
k8s v1.24.3 https://docs.qq.com/doc/DUWRRQmZpeE1Sd1dC
k8s v1.27.1 https://note.youdao.com/s/Dxh3Qaaa

2. K8S 快速实战

2.1 kubectl命令使用

kubectl是apiserver的客户端工具，工作在命令行下，能够连接apiserver实现各种增删改查等操作
kubectl官方使用文档：https://kubernetes.io/zh/docs/reference/kubectl/overview/

K8S的各种命令帮助文档做得非常不错，遇到问题可以多查help帮助

2.2 Namespace

K8s 中，**命名空间（Namespace）** 提供一种机制，将同一集群中的资源划分为相互隔离的组。同一命名空间内的资源名称要唯一，命名空间是用来隔离资源的，不隔离网络。

Kubernetes 启动时会创建四个初始命名空间：

- **default**

Kubernetes 包含这个命名空间，以便于你无需创建新的命名空间即可开始使用新集群。

- **kube-node-lease**

该命名空间包含用于与各个节点关联的 Lease（租约）对象。 节点租约允许 kubelet 发送心跳， 由此控制面能够检测到节点故障。

- **kube-public**

所有的客户端（包括未经身份验证的客户端）都可以读取该命名空间。 该命名空间主要预留为集群使用，以便某些资源需要在整个集群中可见可读。 该命名空间的公共属性只是一种约定而非要求。

- **kube-system**

该命名空间用于 Kubernetes 系统创建的对象。

```
# 查看namespace
kubectl get namespace
#查看kube-system下的pod
kubectl get pods -n kube-system
#查看所有namespace下的pod
kubectl get pods -A
```

创建Namesapce示例

- **命令行方式**

可以使用下面的命令创建Namespace：

```
kubectl create namespace tuling
```

- **yaml方式**

新建一个名为 my-namespace.yaml 的 YAML 文件，并写入下列内容：

```
apiVersion: v1
kind: Namespace
metadata:
  name: tulingmall
```

然后运行：

```
kubectl apply -f my-namespace.yaml
```

删除namesapce

```
kubectl delete namespace tuling

kubectl delete -f my-namespace.yaml
```

2.3 Pod

Pod 是可以在 Kubernetes 中创建和管理的、最小的可部署的计算单元。Pod（就像在鲸鱼荚或者豌豆荚中）**是一组（一个或多个）容器**； 这些容器共享存储、网络、以及怎样运行这些容器的声明。

创建Pod示例：运行一个NGINX容器

- **命令行方式**

```
# 创建pod
kubectl run mynginx --image=nginx:1.14.2
```

```
# 获取pod的信息，-owide 表示更详细的显示信息  -n 命名空间 查询对应namespace下的pod
kubectl get pod
```

```
kubectl get pod -owide
kubectl get pod -owide -n <namespace-name>

#查看pod的详情
kubectl describe pod <pod-name>
# 查看Pod的运行日志
kubectl logs <pod-name>

# 删除pod
kubectl delete pod <pod-name>
```

- **yaml方式**

```
#vim nginx-pod.yaml

apiVersion: v1
kind: Pod
metadata:
  labels:
    run: mynginx
  name: mynginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

然后运行：

```
kubectl apply -f  nginx-pod.yaml
```

删除pod

```
kubectl delete -f  nginx-pod.yaml
```

思考：一个pod中可以运行多个容器吗?

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: myapp
  name: myapp
spec:
  containers:
  - image: nginx:1.14.2
    name: nginx
  - image: tomcat:9.0.55
    name: tomcat
```

2.4 Deployment

Deployment负责创建和更新应用程序的实例，**使Pod拥有多副本，自愈，扩缩容等能力**。创建Deployment后，Kubernetes Master 将应用程序实例调度到集群中的各个节点上。如果托管实例的节点关闭或被删除，Deployment控制器会将该实例替换为群集中另一个节点上的实例。这提供了一种自我修复机制来解决机器故障维护问题。

创建一个Tomcat应用程序

使用 kubectl create deployment 命令可以创建一个应用部署**deployment**与**pod**

```
#my-tomcat表示pod的名称 --image表示镜像的地址
kubectl create deployment my-tomcat --image=tomcat:9.0.55

#查看一下deployment的信息
kubectl get deployment

#删除deployment
```

```
kubectl delete deployment my-tomcat

#查看Pod打印的日志
kubectl logs my-tomcat-6d6b57c8c8-n5gm4

#使用 exec 可以在Pod的容器中执行命令
kubectl exec my-tomcat-6d6b57c8c8-n5gm4 -- env   #使用 env 命令查看环境变量
kubectl exec my-tomcat-6d6b57c8c8-n5gm4 -- ls /   # 查看容器的根目录下面内容
kubectl exec my-tomcat-6d6b57c8c8-n5gm4 -- sh   #进入Pod容器内部并执行bash命令，如果想退出容器可以使用exit命令
```

思考：下面两个命令有什么区别?

```
kubectl create my-tomcat --image=tomcat:9.0.55
kubectl create deployment my-tomcat --image=tomcat:9.0.55
```

自愈

现在我们来删除刚刚添加的pod，看看会发生什么

```
#查看pod信息，-w意思是一直等待观察pod信息的变动
kubectl get pod -w
```

开另外一个命令窗口执行如下命令，同时观察之前命令窗口的变化情况

```
kubectl delete pod  my-tomcat-6d6b57c8c8-n5gm4
```

我们可以看到之前那个tomcat的pod被销毁，但是又重新启动了一个新的tomcat pod，这是k8s的服务**自愈功能**，不需要运维人员干预

多副本

- 命令行的方式

```
# 创建3个副本
kubectl create deployment my-tomcat --image=tomcat:9.0.55 --replicas=3
```

- yaml方式

```

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: my-tomcat
  name: my-tomcat
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-tomcat
  template:
    metadata:
      labels:
        app: my-tomcat
    spec:
      containers:
      - image: tomcat:9.0.55
        name: tomcat
```

扩缩容

```
# 扩容到5个pod
kubectl scale --replicas=5 deployment my-tomcat
# 缩到3个pod
kubectl scale --replicas=3 deployment my-tomcat
```

```
#修改 replicas
kubectl edit deployment my-tomcat
```

滚动升级与回滚

对my-tomcat这个deployment进行滚动升级和回滚，将tomcat版本由tomcat:9.0.55升级到tomcat:10.1.11，再回滚到tomcat:9.0.55

滚动升级：

```
kubectl set image deployment my-tomcat tomcat=tomcat:10.1.11 --record
```

可以执行 kubectl get pod -w 观察pod的变动情况，可以看到有的pod在销毁，有的pod在创建

查看pod信息

```
kubectl get pod
```

查看某个pod的详细信息，发现pod里的镜像版本已经升级了

```
kubectl describe pod my-tomcat-85c5c8f685-lnkfm
```

版本回滚：

查看历史版本

```
kubectl rollout history deploy my-tomcat
```

回滚到上一个版本

```
kubectl rollout undo deployment my-tomcat     #--to-revision 参数可以指定回退的版本

#回滚(回到指定版本)
kubectl rollout undo deployment/my-dep --to-revision=2
```

查看pod详情，发现版本已经回退了

访问tomcat pod

集群内访问（在集群里任一worker节点都可以访问）

```
curl 10.244.169.164:8080
```

集群外部访问

当我们在集群之外访问是发现无法访问，那么集群之外的客户端如何才能访问呢？这就需要我们的service服务了，下面我们就创建一个service，使外部客户端可以访问我们的pod

2.5 Service

Service是一个抽象层，它定义了一组Pod的逻辑集，并为这些Pod支持外部流量暴露、负载均衡和服务发现。

尽管每个Pod 都有一个唯一的IP地址，但是如果没有Service，这些IP不会暴露在群集外部。Service允许您的应用程序接收流量。Service也可以用在ServiceSpec标记type的方式暴露，type类型如下：

- ClusterIP（默认）：在集群的内部IP上公开Service。这种类型使得Service只能从集群内访问。
- NodePort：使用NAT在集群中每个选定Node的相同端口上公开Service。使用 **<NodeIP>:<NodePort>** 从集群外部访问Service。是ClusterIP的超集。
- LoadBalancer：在当前云中创建一个外部负载均衡器(如果支持的话)，并为Service分配一个固定的外部IP。是NodePort的超集。
- ExternalName：通过返回带有该名称的CNAME记录，使用任意名称（由spec中的externalName指定）公开Service。不使用代理。

创建service示例

- 命令行的方式

```
kubectl expose deployment my-tomcat --name=tomcat --port=8080 --type=NodePort

#查看service信息，port信息里冒号后面的端口号就是对集群外暴露的访问接口
# NodePort范围在 30000-32767 之间
kubectl get svc -o wide
```

集群外部访问

使用集群节点的ip加上暴露的端口就可以访问

tomcat版本太高返回404的解决办法：进入tomcat容器，把 webapps 目录删除，再把 webapps.dist 重命名为 webapps 即可。

- yaml的方式

```
# vim mytomcat-service.yaml

apiVersion: v1
kind: Service
metadata:
  labels:
    app: my-tomcat
  name: my-tomcat
spec:
  ports:
  - port: 8080  # service的虚拟ip对应的端口，在集群内网机器可以访问用service的虚拟ip加该端口号访问服务
    nodePort: 30001   # service在宿主机上映射的外网访问端口，端口范围必须在30000-32767之间
    protocol: TCP
    targetPort: 8080  # pod暴露的端口，一般与pod内部容器暴露的端口一致
  selector:
    app: my-tomcat
  type: NodePort

```

执行如下命令创建service：

```
kubectl apply -f mytomcat-service.yaml
```

集群外部访问

2.6 存储

Volume

Volume指的是存储卷，包含可被Pod中容器访问的数据目录。容器中的文件在磁盘上是临时存放的，当容器崩溃时文件会丢失，同时无法在多个Pod中共享文件，通过使用存储卷可以解决这两个问题。Kubernetes 支持很多类型的卷。 Pod 可以同时使用任意数目的卷类型。 临时卷类型的生命周期与 Pod 相同，但持久卷可以比 Pod 的存活期长。 当 Pod 不再存在时，Kubernetes 也会销毁临时卷；不过 Kubernetes 不会销毁持久卷。 对于给定 Pod 中任何类型的卷，在容器重启期间数据都不会丢失。

卷的核心是一个目录，其中可能存有数据，Pod 中的容器可以访问该目录中的数据。 所采用的不同卷的类型将决定该目录如何形成的、使用何种介质保存数据以及目录中存放的内容。常用的卷类型有 **configMap、emptyDir、local、nfs、secret** 等。

- ConfigMap：可以将配置文件以键值对的形式保存到 ConfigMap 中，并且可以在 Pod 中以文件或环境变量的形式使用。ConfigMap 可以用来存储不敏感的配置信息，如应用程序的配置文件。
- EmptyDir：是一个空目录，可以在 Pod 中用来存储临时数据，当 Pod 被删除时，该目录也会被删除。
- Local：将本地文件系统的目录或文件映射到 Pod 中的一个 Volume 中，可以用来在 Pod 中共享文件或数据。
- NFS：将网络上的一个或多个 NFS 共享目录挂载到 Pod 中的 Volume 中，可以用来在多个 Pod 之间共享数据。
- Secret：将敏感信息以密文的形式保存到 Secret 中，并且可以在 Pod 中以文件或环境变量的形式使用。Secret 可以用来存储敏感信息，如用户名密码、证书等。

使用方式

使用卷时, 在 .spec.volumes 字段中设置为 Pod 提供的卷，并在 .spec.containers[*].volumeMounts 字段中声明卷在容器中的挂载位置。 容器中的进程看到的文件系统视图是由它们的容器镜像的初始内容以及挂载在容器中的卷（如果定义了的话）所组成的。 其中根文件系统同容器镜像的内容相吻合。

任何在该文件系统下的写入操作，如果被允许的话，都会影响接下来容器中进程访问文件系统时所看到的内容。

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod
spec:
  containers:
    - name: test
      image: busybox:1.28
      volumeMounts:
                ..........
  volumes:
        ............
```

搭建nfs文件系统

nfs(network filesystem): 网络文件存储系统

安装nfs-server

```
# 在每个机器。
yum install -y nfs-utils

# 在master 执行以下命令
echo "/nfs/data/ *(insecure,rw,sync,no_root_squash)" > /etc/exports

# 执行以下命令，启动 nfs 服务;创建共享目录
mkdir -p /nfs/data

# 在master执行
systemctl enable rpcbind
systemctl enable nfs-server
systemctl start rpcbind
systemctl start nfs-server

# 使配置生效
```

```
exportfs -r

#检查配置是否生效
exportfs
```

配置nfs-client

```
# nfs-server节点的ip
showmount -e 192.168.11.101
mkdir -p /nfs/data
mount -t nfs 192.168.11.101:/nfs/data /nfs/data

```

nfs方式数据挂载

相对于 emptyDir 和 hostPath，nfs这种 Volume 类型的最大特点就是不依赖 Kuberees Volume 的底层基础设施由独立的存储系统管理，与 Kubernetes 集群是分离的。数据被持久化后，即使整个 Kubernetes 崩溃也不会受损。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx-pv-demo
  name: nginx-pv-demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx-pv-demo
  template:
    metadata:
      labels:
        app: nginx-pv-demo
    spec:
      containers:
      - image: nginx
```

```
        name: nginx
        volumeMounts:
        - name: html
          mountPath: /usr/share/nginx/html
      volumes:
        - name: html
          nfs:
            server: 192.168.11.101
            path: /nfs/data/nginx-pv
```

PV & PVC

Volume 提供了非常好的数据持久化方案，不过在可管理性上还有不足。前面 nfs 例子来说，要使用 Volume， Pod 必须事先知道以下信息：

- 当前的 Volume 类型并明确 Volume 已经创建好。
- 必须知道 Volume 的具体地址信息。

但是 Pod 通常是由应用的开发人员维护，而 Volume 则通常是由存储系统的管理员维护。开发人员要获得上面的信息，要么询问管理员，要么自己就是管理员。这样就带来一个管理上的问题：应用开发人员和系统管理员的职责耦合在一起了。如果系统规模较小或者对于开发环境，这样的情况还可以接受，当集群规模变大，特别是对于生产环境，考虑到效率和安全性，这就成了必须要解决的问题。

Kubernetes 给出的解决方案是 Persistent Volume 和 Persistent Volume Claim。

PersistentVolume(PV）是外部存储系统中的一块存储空间，由管理员创建和维护，**将应用需要持久化的数据保存到指定位置**。与 Volume 一样，PV 具有持久性，生命周期独立于 Pod。

Persistent Volume Claim (PVC)是对 PV 的申请 (Claim），**申明需要使用的持久卷规格**。PVC 通常由普通用户创建和维护。需要为 Pod 分配存储资源时，用户可以创建一个PVC，指明存储资源的容量大小和访问模式 （比如只读）等信息，Kubernetes 会查找并提供满足条件的 PV。有了 PersistentVolumeClaim，用户只需要告诉 Kubernetes 需要什么样的存储资源，而不必关心真正的空间从哪里分配、如何访问等底层细节信息。这些 Storage Provider 的底层信息交给管理员来处理，只有管理员才应该关心创建 PersistentVolume 的细节信息。

基本使用

创建pv

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv
```

```
spec:
  capacity:
    storage: 1Gi #指定容量大小
  accessModes: # 访问模式
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  storageClassName: nfs
  nfs:
    path: /{nfs-server目录名称}
    server: {nfs-server IP 地址}
```

- accessModes: 支持的访问模式有3种：
  - ReadWriteOnce 表示 PV 能以 readwrite 模式 mount 到单个节点
    - 这个PV只能被某个节点以读写方式挂载，意味着这个PV只能被一个Pod挂载到某个节点上，并且这个Pod可以对这个PV进行读写操作。如果尝试在其他节点上挂载这个PV，就会失败。
  - ReadOnlyMany 表示 PV 能以 read-only 模式 mount 到多个节点，
    - 这个PV能被多个节点以只读方式挂载，意味着这个PV可以被多个Pod挂载到多个节点上。
  - ReadWriteMany 表示 PV 能以 read-write 模式 mount 到多个节点。
    - 这个PV能被多个节点以读写方式挂载，意味着这个PV可以被多个Pod挂载到多个节点上。
- persistentVolumeReclaimPolicy: 指定当 PV 的回收策略支持的策略有3种：
  - Retain：在 PVC 被删除后，保留 PV 和其数据，手动清理 PV 中的数据。
  - Delete：在 PVC 被删除后，自动删除 PV 和其数据。
  - Recycle：在 PVC 被删除后，通过删除 PV 中的数据来准备 PV 以供重新使用。

值得注意的是，persistentVolumeReclaimPolicy只适用于一些类型的 PV，如 NFS、HostPath、iSCSI 等。对于一些云平台提供的存储，如 AWS EBS 和 Azure Disk，由于底层提供商会自动处理 PV 的回收问题，因此该属性不适用。

- storageClassName: 指定 PV 的class 为 nfs。相当于为 PV 设置了一个分类，PVC可以指定 class 申请相应 class 的 PV。

创建pvc

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  accessModes:
    - ReadWriteMany
```

```
  resources:
    requests:
      storage: 1Gi
  storageClassName: nfs # 通过名字进行选择
  #selector:  通过标签形式
  #  matchLabels:
  #    pv-name: nfs-pv
```

静态供应示例

创建PV池

```
#nfs主节点
mkdir -p /nfs/data/01
mkdir -p /nfs/data/02
mkdir -p /nfs/data/03
```

创建PV

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv01-10m
spec:
  capacity:
    storage: 10M
  accessModes:
    - ReadWriteMany
  storageClassName: nfs
  nfs:
    path: /nfs/data/01
    server: 192.168.11.101
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv02-1gi
```

```yaml
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
  storageClassName: nfs
  nfs:
    path: /nfs/data/02
    server: 192.168.11.101
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv03-3gi
spec:
  capacity:
    storage: 3Gi
  accessModes:
    - ReadWriteMany
  storageClassName: nfs
  nfs:
    path: /nfs/data/03
    server: 192.168.11.101
```

创建PVC

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nginx-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 200Mi
  storageClassName: nfs
```

创建Pod绑定PVC

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx-deploy-pvc
  name: nginx-deploy-pvc
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx-deploy-pvc
  template:
    metadata:
      labels:
        app: nginx-deploy-pvc
    spec:
      containers:
      - image: nginx
        name: nginx
        volumeMounts:
        - name: html
          mountPath: /usr/share/nginx/html
      volumes:
        - name: html
          persistentVolumeClaim:
            claimName: nginx-pvc
```

动态供应

在前面的例子中，我们提前创建了PV，然后通过 PVC 申请 PV 并在Pod 中使用，这种方式叫作静态供应（Static Provision)与之对应的是动态供应 (Dynamical Provision），即如果没有满足PVC 条件的 PV，会动态创建 PV。相比静态供应，动态供应有明显的优势：不需要提前创建 PV，减少了管理员的工作量，效率高。动态供应是通过 StorageClass 实现的，StorageClass 定义了如何创建 PV，但需要注意的是每个 StorageClass 都有一个制备器（Provisioner），用来决定使用哪个卷插件制备 PV，该字段必须指定才能实现动态创建。

配置动态供应的默认存储类

```yaml
## 创建了一个存储类
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-storage
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: k8s-sigs.io/nfs-subdir-external-provisioner
parameters:
  archiveOnDelete: "true"  ## 删除pv的时候，pv的内容是否要备份

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nfs-client-provisioner
  labels:
    app: nfs-client-provisioner
  # replace with namespace where provisioner is deployed
  namespace: default
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: nfs-client-provisioner
  template:
    metadata:
      labels:
        app: nfs-client-provisioner
    spec:
      serviceAccountName: nfs-client-provisioner
      containers:
        - name: nfs-client-provisioner
          image: registry.cn-hangzhou.aliyuncs.com/lfy_k8s_images/nfs-subdir-external-provisioner:v4.0.2
```

```
          # resources:
          #    limits:
          #      cpu: 10m
          #    requests:
          #      cpu: 10m
          volumeMounts:
            - name: nfs-client-root
              mountPath: /persistentvolumes
          env:
            - name: PROVISIONER_NAME
              value: k8s-sigs.io/nfs-subdir-external-provisioner
            - name: NFS_SERVER
              value: 192.168.11.101 ## 指定自己nfs服务器地址
            - name: NFS_PATH
              value: /nfs/data  ## nfs服务器共享的目录
      volumes:
        - name: nfs-client-root
          nfs:
            server: 192.168.11.101
            path: /nfs/data
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: nfs-client-provisioner
  # replace with namespace where provisioner is deployed
  namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: nfs-client-provisioner-runner
rules:
  - apiGroups: [""]
    resources: ["nodes"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["persistentvolumes"]
    verbs: ["get", "list", "watch", "create", "delete"]
  - apiGroups: [""]
```

```
    resources: ["persistentvolumeclaims"]
    verbs: ["get", "list", "watch", "update"]
  - apiGroups: ["storage.k8s.io"]
    resources: ["storageclasses"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["events"]
    verbs: ["create", "update", "patch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: run-nfs-client-provisioner
subjects:
  - kind: ServiceAccount
    name: nfs-client-provisioner
    # replace with namespace where provisioner is deployed
    namespace: default
roleRef:
  kind: ClusterRole
  name: nfs-client-provisioner-runner
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: leader-locking-nfs-client-provisioner
  # replace with namespace where provisioner is deployed
  namespace: default
rules:
  - apiGroups: [""]
    resources: ["endpoints"]
    verbs: ["get", "list", "watch", "create", "update", "patch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: leader-locking-nfs-client-provisioner
  # replace with namespace where provisioner is deployed
```

```
  namespace: default
subjects:
  - kind: ServiceAccount
    name: nfs-client-provisioner
    # replace with namespace where provisioner is deployed
    namespace: default
roleRef:
  kind: Role
  name: leader-locking-nfs-client-provisioner
  apiGroup: rbac.authorization.k8s.io
```

2.7 配置

ConfigMap

在 Kubernetes 中，**ConfigMap 是一种用于存储非敏感信息的 Kubernetes 对象。**它用于存储配置数据，如键值对、整个配置文件或 JSON 数据等。ConfigMap 通常用于容器镜像中的配置文件、命令行参数和环境变量等。

ConfigMap 可以通过三种方式进行配置数据的注入：

1. 环境变量注入：将配置数据注入到 Pod 中的容器环境变量中。
2. 配置文件注入：将配置数据注入到 Pod 中的容器文件系统中，容器可以读取这些文件。
3. 命令行参数注入：将配置数据注入到容器的命令行参数中。

优点

1. 避免了硬编码，将配置数据与应用代码分离。
2. 便于维护和更新，可以单独修改 ConfigMap 而不需要重新构建镜像。
3. 可以通过多种方式注入配置数据，更加灵活。
4. 可以通过 Kubernetes 的自动化机制对 ConfigMap 进行版本控制和回滚。
5. ConfigMap 可以被多个 Pod 共享，减少了配置数据的重复存储。

定义 ConfigMap

- 基本操作

```
# 查看 configmap
$ kubectl get configmap/cm
```

```
3
4 # 查看详细
5 $ kubectl describe configmap/cm my-config
6
7 # 删除 cm
8 $ kubectl delete cm my-config
```

- 命令行创建：
  - 可以使用kubectl create configmap命令来创建configmap，具体命令如下：

```
1  kubectl create configmap my-config --from-literal=key1=value1 --from-
literal=key2=value2
```

- 通过配置文件创建：**推荐**
  - 可以通过创建YAML文件的方式来定义configmap的内容。例如，创建一个名为my-config的configmap，内容如下：

```
1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: my-config
5 data:
6   key1: value1
7   key2: value2
8
9 ---
10 apiVersion: v1
11 kind: ConfigMap
12 metadata:
13   name: app-config
14 data:
15   application.yml: |
16     name: fox
```

  - 然后使用kubectl apply -f命令来创建configmap。
- 通过文件创建:

```
echo-n admin >./username
echo -n 123456 > ./password
kubectl create configmap myconfigmap --from-file=./username --from-file=./password
```

- 通过文件夹创建：
  - 可以将多个配置文件放在同一个文件夹下，然后使用kubectl create configmap命令来创建configmap，例如：

```
 kubectl create configmap my-config --from-file=config-files/
```

  - 这将创建一个名为my-config的configmap，其中包含config-files/文件夹下所有的文件内容作为键值对。
- 通过环境变量创建：
  - 可以将环境变量的值转换为configmap。例如，使用以下命令将当前环境变量的值转换为configmap：

```
 kubectl create configmap my-config --from-env-file=<env>
```

使用示例

```
# docker安装redis
docker run -v /data/redis/redis.conf:/etc/redis/redis.conf \
-v /data/redis/data:/data \
-d --name myredis \
-p 6379:6379 \
redis:latest  redis-server /etc/redis/redis.conf
```

创建ConfigMap

- 通过文件的方式创建

```
#创建redis.conf
daemonize yes

requirepass root

```

```
# 创建配置，redis保存到k8s的etcd
kubectl create cm redis-conf --from-file=redis.conf

#查看资源清单
kubectl get cm redis-conf -oyaml
```

- 通过yaml的方式创建

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: redis-conf
data:
  redis.conf: |
    maxmemory-policy allkeys-lru
    requirepass root
```

创建Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    command:
      - redis-server
      - "/redis-master/redis.conf"  #指的是redis容器内部的位置
    ports:
    - containerPort: 6379
    volumeMounts:
    - mountPath: /data
      name: data
```

```
    - mountPath: /redis-master
      name: config
  volumes:
    - name: data
      emptyDir: {}
    - name: config
      configMap:
        name: redis-conf
        items:
        - key: redis.conf
          path: redis.conf
```

测试

```
kubectl exec -it redis -- redis-cli
127.0.0.1:6379> config get  maxmemory-policy

```

Secret

Secret 对象类型用来保存敏感信息，例如密码、OAuth 令牌和 SSH 密钥。 将这些信息放在 secret 中比放在 Pod 的定义或者 容器镜像 中来说更加安全和灵活。

在 Kubernetes 中，Secrets 通常被用于以下场景：

- 作为卷挂载到 Pod 中，用于存储证书、密钥等敏感文件
- 在 Pod 中使用环境变量，用于存储用户名和密码等敏感信息
- 用于存储 Docker 镜像仓库的登录信息
- 用于存储外部服务的 API 密钥

定义 Secret

- 使用命令行创建：
  - 可以使用 kubectl create secret 命令来创建 secret，例如：

```
kubectl create secret generic my-secret --from-literal=username=admin --from-
literal=password=admin123
```

- 使用 YAML 文件定义：
  - 可以创建一个 YAML 文件来定义 Secret 对象，例如：

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: YWRtaW4= # base64 编码后的用户名 admin
  password: MWYyZDFlMmU2N2Rm # base64 编码后的密码 1f2d1e2e67df
```

注意: 这个 YAML 文件定义了一个名为 my-secret 的 Secret 对象，其中包含了两个 base64 编码后的 key-value 对：username 和 password。

- 使用文件创建:

```
echo -n admin >./username
echo -n 123456 > ./password
kubectl create secret generic mysecret --from-file=./username --from-file=./password
```

- 通过环境变量创建：
  - 可以将环境变量的值转换为secret。例如，使用以下命令将当前环境变量的值转换为secret：

```
kubectl create secret generic  my-config --from-env-file=<env>
```

使用示例：从私有docker仓库拉取镜像

```
docker pull registry.cn-hangzhou.aliyuncs.com/fox666/tulingmall-product:0.0.5
```

无法从私有镜像仓库拉取镜像，抛出如下错误：

解决方案：使用 docker 的用户信息来生成 secret：

```
##命令格式
kubectl create secret docker-registry myregistrykey \
  --docker-server=<你的镜像仓库服务器> \
  --docker-username=<你的用户名> \
  --docker-password=<你的密码> \
  --docker-email=<你的邮箱地址>

  kubectl create secret docker-registry myregistrykey --docker-server=registry.cn-hangzhou.aliyuncs.com --docker-username=fox666 --docker-password=xxx
```

在创建 Pod 的时候，通过imagePullSecrets来引用刚创建的myregistrykey

```
apiVersion: v1
kind: Pod
metadata:
  name: tulingmall-product
spec:
  containers:
  - name: tulingmall-product
    image: registry.cn-hangzhou.aliyuncs.com/fox666/tulingmall-product:0.0.5
  imagePullSecrets:
  - name: myregistrykey
```

2.8 Ingress

Ingress 是一种 Kubernetes 资源类型，它允许在 Kubernetes 集群中暴露 HTTP 和 HTTPS 服务。
通过 Ingress，您可以将流量路由到不同的服务和端点，而无需使用不同的负载均衡器。Ingress 通常使用 Ingress Controller 实现，它是一个运行在 Kubernetes 集群中的负载均衡器，它根据Ingress 规则配置路由规则并将流量转发到相应的服务。
下面是一个将所有流量都发送到同一 Service 的简单 Ingress 示例：

Ingress 和 Service 区别

Ingress 和 Service都是 Kubernetes 中用于将流量路由到应用程序的机制，但它们在路由层面上有所不同：

- Service 是 Kubernetes 中抽象的应用程序服务，它公开了一个单一的IP地址和端口，可以用于在 Kubernetes

集群内部的 Pod 之间进行流量路由。

- Ingress 是一个 Kubernetes 资源对象，它提供了对集群外部流量路由的规则。Ingress 通过一个公共IP地址和端口将流量路由到一个或多个Service。

安装Ingress

1）下载ingress配置文件

```
wget https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v0.47.0/deploy/static/provider/baremetal/deploy.yaml


[root@k8s-master k8s]# grep image: deploy.yaml
          image: k8s.gcr.io/ingress-nginx/controller:v0.46.0@sha256:52f0058bed0a17ab0fb35628ba97e8d52b5d32299fbc03cc0f6c7b9ff036b61a
          image: docker.io/jettech/kube-webhook-certgen:v1.5.1
          image: docker.io/jettech/kube-webhook-certgen:v1.5.1

#修改镜像
vi deploy.yaml
#1、将image k8s.gcr.io/ingress-nginx/controller:v0.46.0@sha256:52f0058bed0a17ab0fb35628ba97e8d52b5d32299fbc03cc0f6c7b9ff036b61a的值改为如下值：
registry.cn-hangzhou.aliyuncs.com/lfy_k8s_images/ingress-nginx-controller:v0.46.0
```

2）安装ingress，执行如下命令

```
kubectl apply -f ingress-controller.yaml
```

3) 查看是否安装成功

```
kubectl get pod,svc -n ingress-nginx -owide
```

使用Ingress

官网地址：https://kubernetes.github.io/ingress-nginx/

配置ingress访问规则（就是类似配置nginx的代理转发配置），让ingress将域名tomcat.tuling.com转发给后端的my-tomcat服务，新建一个文件ingress-tomcat.yaml，内容如下：

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: web-ingress
spec:
  rules:
  - host: tomcat.tuling.com  #转发域名
    http:
      paths:
      - pathType: Prefix
        path: /
        backend:
          service:
            name: my-tomcat
            port:
              number: 8080  #service的端口
```

执行如下命令生效规则：

```
kubectl apply -f ingress-tomcat.yaml
```

查看生效的ingress规则：

```
kubectl get ing
```

在访问机器配置host，win10客户机在目录：C:\Windows\System32\drivers\etc，在host里增加如下host(ingress部署的机器ip对应访问的域名)

```
192.168.65.82  tomcat.tuling.com
```

配置完后直接在客户机浏览器访问http://tomcat.tuling.com:30940能正常访问tomcat。

Service&Ingress总结

Service 是 K8S 服务的核心，屏蔽了服务细节，统一对外暴露服务接口，真正做到了“微服务”。举个例子，我们的一个服务 A，部署了 3 个备份，也就是 3 个 Pod；对于用户来说，只需要关注一个 Service 的入口就可以，而不需要操心究竟应该请求哪一个 Pod。优势非常明显：**一方面外部用户不需要感知因为 Pod 上服务的意外崩溃、K8S 重新拉起 Pod 而造成的 IP 变更，外部用户也不需要感知因升级、变更服务带来的 Pod 替换而造成的 IP 变化，另一方面，Service 还可以做流量负载均衡。**
但是，Service 主要负责 K8S 集群**内部的网络拓扑**。集群外部需要用 Ingress 。
Ingress 是整个 K8S 集群的接入层，复杂集群内外通讯。
Ingress 和 Service 的网络拓扑关系图如下：

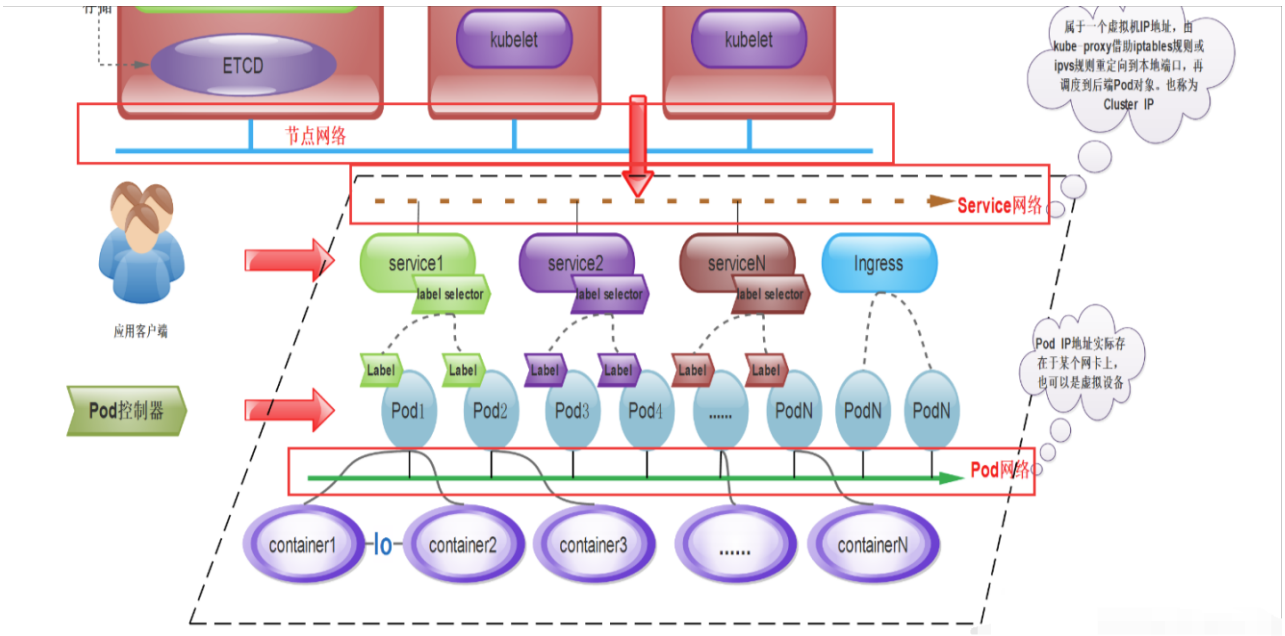
3. K8S核心原理

3.1 K8S的网络模型

K8S的网络中主要存在4种类型的通信：

- 同一Pod内的容器间通信
- 各个Pod彼此间的通信
- Pod和Service间的通信
- 集群外部流量和Service之间的通信

K8S为Pod和Service资源对象分别使用了各自的专有网络，Pod网络由K8S的网络插件配置实现，而Service网络则由K8S集群进行指定。如下图：



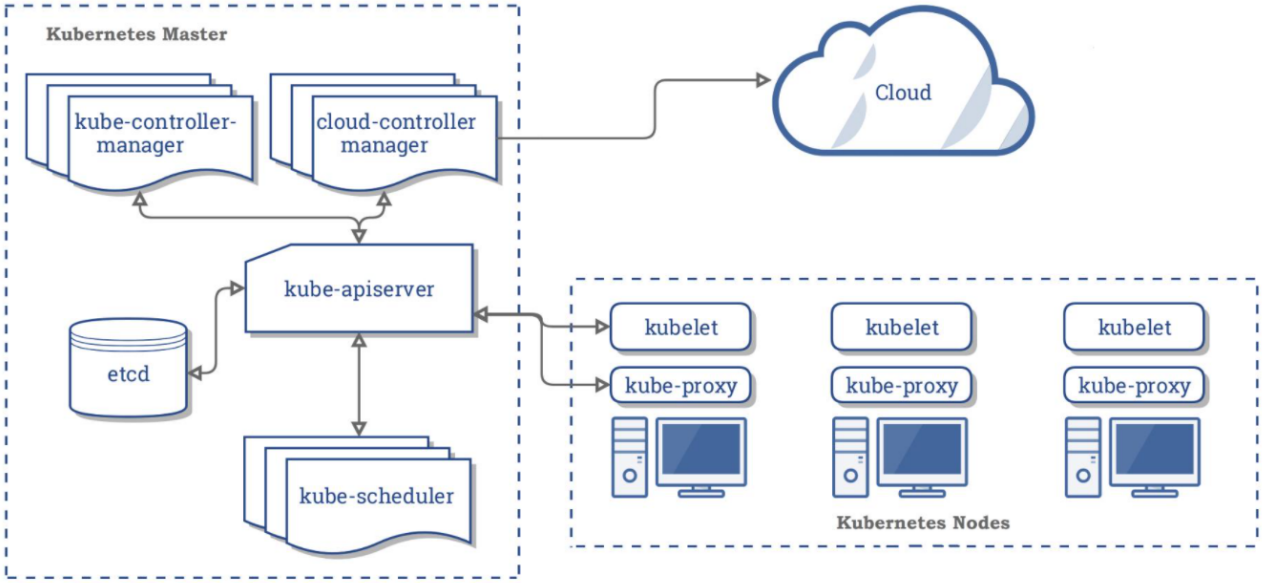
K8S使用的网络插件需要为每个Pod配置至少一个特定的地址，即Pod IP。Pod IP地址实际存在于某个网卡（可以是虚拟机设备）上。
而Service的地址却是一个虚拟IP地址，没有任何网络接口配置在此地址上，它由Kube-proxy借助iptables规则或ipvs规则重定向到本地端口，再将其调度到后端的Pod对象。Service的IP地址是集群提供服务的接口，也称为Cluster IP。
Pod网络和IP由K8S的网络插件负责配置和管理，具体使用的网络地址可以在管理配置网络插件时进行指定，如10.244.0.0/16网络。而Cluster网络和IP是由K8S集群负责配置和管理，如10.96.0.0/12网络。
从上图进行总结起来，一个K8S集群包含是三个网络。

- 节点网络：各主机（Master、Node、ETCD等）自身所属的网络，地址配置在主机的网络接口，用于各主机之间的通信，又称为节点网络。
- Pod网络：专用于Pod资源对象的网络，它是一个虚拟网络，用于为各Pod对象设定IP地址等网络参数，其地址配置在Pod中容器的网络接口上。Pod网络需要借助kubenet插件或CNI插件实现。
- Service网络：专用于Service资源对象的网络，它也是一个虚拟网络，用于为K8S集群之中的Service配置IP地址，但是该地址不会配置在任何主机或容器的网络接口上，而是通过Node上的kube-proxy配置为iptables或ipvs规则，从而将发往该地址的所有流量调度到后端的各Pod对象之上。

3.2 K8S的工作流程

用K8S部署Nginx的过程中，K8S内部各组件是如何协同工作的：
我们在master节点执行一条命令要master部署一个nginx应用(kubectl create deployment nginx --image=nginx)

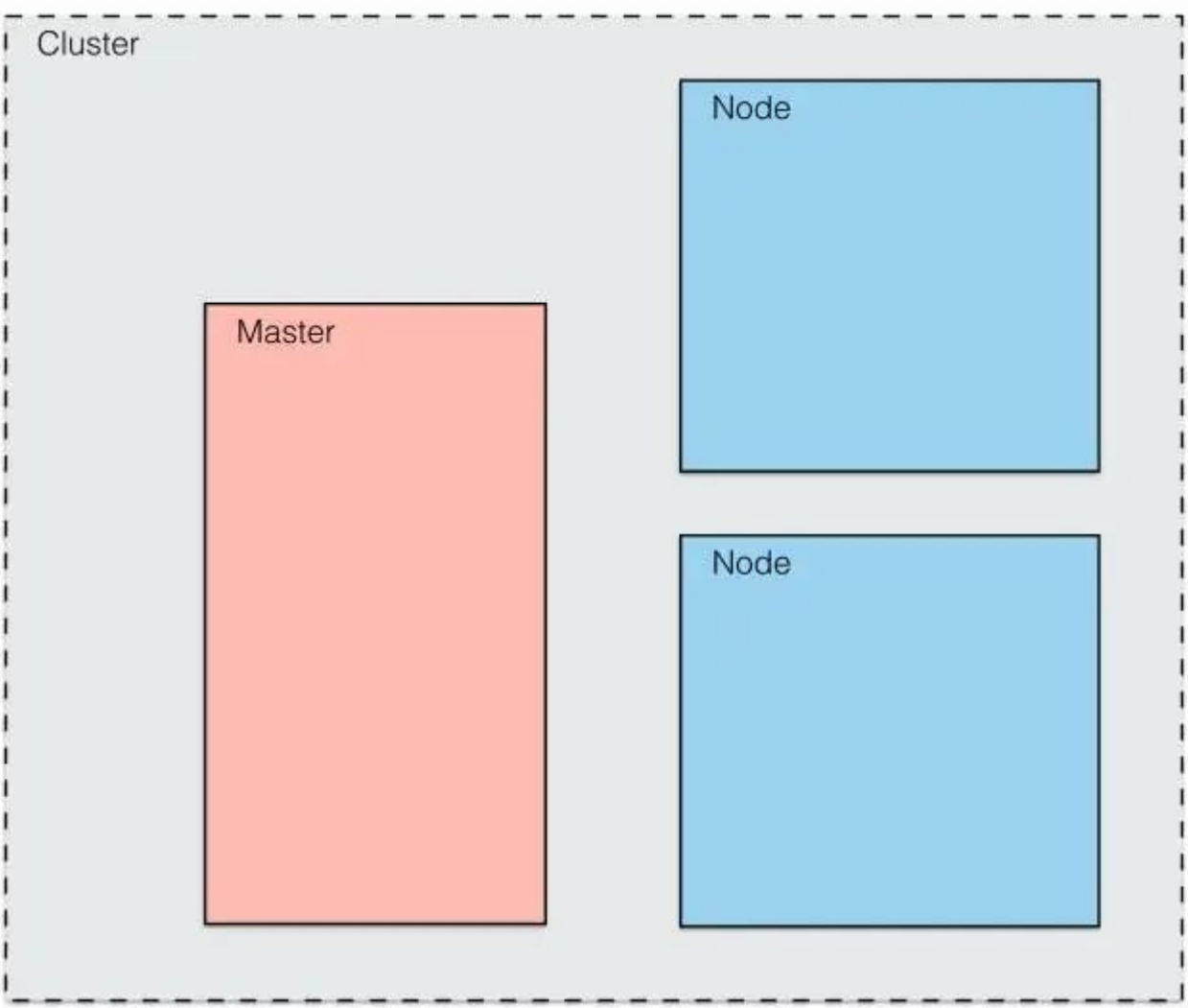
- 这条命令首先发到master节点的网关api server，这是matser的唯一入口
- api server将命令请求交给controller mannager进行控制
- controller mannager 进行应用部署解析
- controller mannager 会生成一次部署信息，并通过api server将信息存入etcd存储中
- scheduler调度器通过api server从etcd存储中，拿到要部署的应用，开始调度看哪个节点有资源适合部署
- scheduler把计算出来的调度信息通过api server再放到etcd中
- 每一个node节点的监控组件kubelet，随时和master保持联系（给api-server发送请求不断获取最新数据），拿到master节点存储在etcd中的部署信息
- 假设node2的kubelet拿到部署信息，显示他自己节点要部署某某应用
- kubelet就自己run一个应用在当前机器上，并随时给master汇报当前应用的状态信息
- node和master也是通过master的api-server组件联系的
- 每一个机器上的kube-proxy能知道集群的所有网络，只要node访问别人或者别人访问node，node上的kube-proxy网络代理自动计算进行流量转发



3.3 k8s架构原理六连问

K8S 是一个基于容器技术的分布式集群管理系统。既然是个分布式系统，那势必有多个 Node 节点（物理主机或虚拟机），它们共同组成一个分布式集群，并且这些节点中会有一个 Master 节点，由它来统一管理 Node 节点。

如图所示：



问题一：主节点和工作节点是如何通信的呢?

首先，Master 节点启动时，会运行一个 kube-apiserver 进程，它提供了集群管理的 API 接口，是集群内各个功能模块之间数据交互和通信的中心枢纽，并且它页提供了完备的集群安全机制。

在 Node 节点上，使用 K8S 中的 kubelet 组件，在每个 Node 节点上都会运行一个 kubelet 进程，它负责向 Master 汇报自身节点的运行情况，如 Node 节点的注册、终止、定时上报健康状况等，以及接收 Master 发出的命令，创建相应 Pod。

在 K8S 中，Pod 是最基本的操作单元，它与 docker 的容器有略微的不同，因为 Pod 可能包含一个或多个容器（可以是 docker 容器），这些内部的容器是共享网络资源的，即可以通过 localhost 进行相互访问。

关于 Pod 内是如何做到网络共享的，每个 Pod 启动，内部都会启动一个 pause 容器（google的一个镜像），它使用默认的网络模式，而其他容器的网络都设置给它，以此来完成网络的共享问题。

如图所示:

问题二：Master 是如何将 Pod 调度到指定的 Node 上的?

该工作由 kube-scheduler 来完成，整个调度过程通过执行一些列复杂的算法最终为每个 Pod 计算出一个最佳的目标 Node，该过程由 kube-scheduler 进程自动完成。常见的有轮询调度（RR）。当然也有可能，我

们需要将 Pod 调度到一个指定的 Node 上，我们可以通过节点的标签（Label）和 Pod 的 nodeSelector 属性的相互匹配，来达到指定的效果。
如图所示：

问题三：各节点、Pod 的信息都是统一维护在哪里的，由谁来维护?

从上面的 Pod 调度的角度看，我们得有一个存储中心，用来存储各节点资源使用情况、健康状态、以及各 Pod 的基本信息等，这样 Pod 的调度来能正常进行。
在 K8S 中，采用 etcd 组件 作为一个高可用强一致性的存储仓库，该组件可以内置在 K8S 中，也可以外部搭建供 K8S 使用。
集群上的所有配置信息都存储在了 etcd，为了考虑各个组件的相对独立，以及整体的维护性，对于这些存储数据的增、删、改、查，统一由 kube-apiserver 来进行调用，apiserver 也提供了 REST 的支持，不仅对各个内部组件提供服务外，还对集群外部用户暴露服务。
外部用户可以通过 REST 接口，或者 kubectl 命令行工具进行集群管理，其内在都是与 apiserver 进行通信。
如图所示：

问题四：外部用户如何访问集群内运行的 Pod？

前面讲了外部用户如何管理 K8S，而我们更关心的是内部运行的 Pod 如何对外访问。使用过Docker 的同学应该知道，如果使用 bridge 模式，在容器创建时，都会分配一个虚拟 IP，该 IP 外部是没法访问到的，我们需要做一层端口映射，将容器内端口与宿主机端口进行映射绑定，这样外部通过访问宿主机的指定端口，就可以访问到内部容器端口了。
那么，K8S 的外部访问是否也是这样实现的？答案是否定的，K8S 中情况要复杂一些。因为上面讲的 Docker 是单机模式下的，而且一个容器对外就暴露一个服务。在分布式集群下，一个服务往往由多个 Application 提供，用来分担访问压力，而且这些 Application 可能会分布在多个节点上，这样又涉及到了跨主机的通信。
这里，K8S 引入了 Service 的概念，将多个相同的 Pod 包装成一个完整的 service 对外提供服务，至于获取到这些相同的 Pod，每个 Pod 启动时都会设置 labels 属性，在 Service 中我们通过选择器 Selector，选择具有相同 Name 标签属性的 Pod，作为整体服务，并将服务信息通过 Apiserver 存入 etcd 中，该工作由 Service Controller 来完成。同时，每个节点上会启动一个 kube-proxy 进程，由它来负责服务地址到 Pod 地址的代理以及负载均衡等工作。
如图所示：

问题五：Pod 如何动态扩容和缩放?

既然知道了服务是由 Pod 组成的，那么服务的扩容也就意味着 Pod 的扩容。通俗点讲，就是在需要时将 Pod 复制多份，在不需要后，将 Pod 缩减至指定份数。K8S 中通过 Replication Controller 来进行管理，为每个 Pod 设置一个期望的副本数，当实际副本数与期望不符时，就动态的进行数量调整，以达到期望值。期望数值可以由我们手动更新，或自动扩容代理来完成。
如图所示：

问题六：各个组件之间是如何相互协作的？

最后，讲一下 kube-controller-manager 这个进程的作用。我们知道了 ectd 是作为集群数据的存储中心，apiserver 是管理数据中心，作为其他进程与数据中心通信的桥梁。而 Service Controller、Replication Controller 这些统一交由 kube-controller-manager 来管理，kube-controller-manager 作为一个守护进程，每个 Controller 都是一个控制循环，通过 apiserver 监视集群的共享状态，并尝试将实际状态与期望不符的进行改变。关于 Controller，manager 中还包含了 Node 节点控制器（Node Controller）、资源配额管控制器（ResourceQuota Controller）、命名空间控制器（Namespace Controller）等。
如图所示：