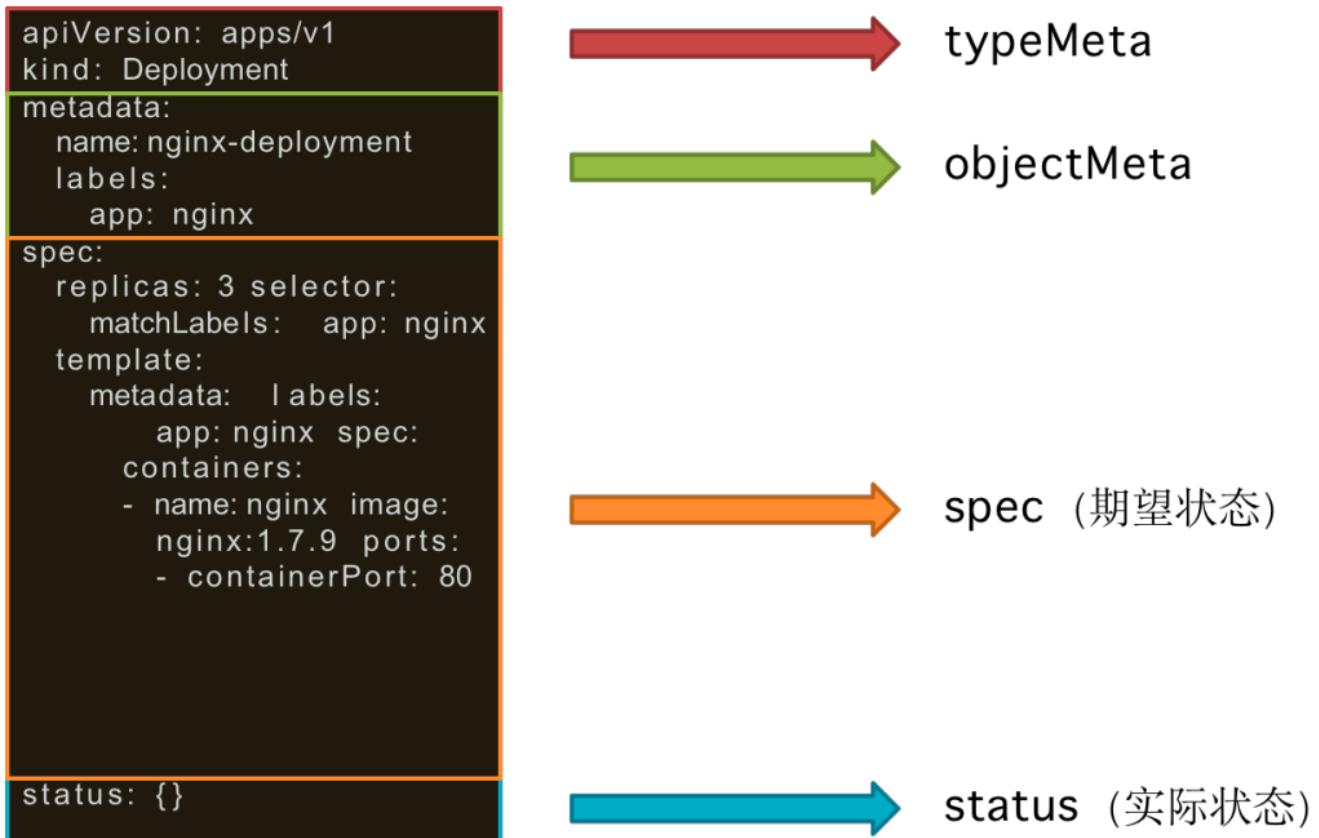


核心概念

- **pod**
最小调度单元，提供容器运行环境，定义容器执行方式
- **Volume**
Pod可访问的文件目录；支持多种存储抽象
- **Deployment**
管理Pod部署的副本、部署方案版本
- **Service**
提供Pod对外访问地址
- **Namespace**
集群内资源隔离的机制

Yaml文件格式：



Pod是k8s的逻辑概念，Pod类似为进程组，Pod中的container类似为进程，一个pod中的所有进程共享资源、紧密协作。容器本身是单进程模型，该进程就是应用进程。这样从外部就可以知道容器中的应用状态。容器如果是多进程，由一个主进程管理其它进程时，管理容器等于管理主进程而非应用本身。管理应用的生命周期会变得困难。

容器资源控制

支持管理的资源包括：CPU、Memory、ephemeral storage（临时存储）及扩展资源如GPU。扩展资源只能获取整数个资源。

containers下面resource中进行资源的request和limits配置

```

apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  containers:
    - name: test
      image: busybox
      resources:
        request:
          memory: "64Mi"
          cpu: "250m"
          ephemeral-storage: "2Gi"
        limits:
          memory: "128Mi"
          cpu: "500m"

```

```
ephemeral-storage: "4Gi"
```

Pod QoS服务质量

对 pod 的服务质量进行一个分类，分别是 Guaranteed、Burstable 和 BestEffort。

- Guaranteed：pod 里面每个容器都必须有内存和 CPU 的 request 以及 limit 的一个声明，且 request 和 limit 必须是一样的，这就是 Guaranteed；保障资源
- Burstable：至少有一个容器存在内存和 CPU 的一个 request；弹性保证
- BestEffort：request/limit 都不填；尽力而为

系统时用request进行调度的。cpu资源而言，系统会对Guaranteed的pod单独划分cpu资源，Burstable和BestEffort则共用cpu，按权重来使用不同的时间片。内存资源而言，会对pod划分OOMScore，score越高越先被kill掉。Guaranteed为固定-998，BestEffort为固定1000，Burstable则是根据内存大小和节点关系划分2~999。发生驱逐时，优先BestEffort。

资源Quota

限制每个Namespace资源用量，超出限制时，用户无法提交新建。

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: demo
  namespace: demo
spec:
  hard:
    cpu: "1000"
    memory: 200Gi
    pods: "10"
```

亲和度调度

Pod亲和调度

1. PodAffinity

- requireDuringSchedulingIgnoredDuringExecution：强制亲和（调度时，执行亲和校验；运行时，若标签发生变化导致的亲和变化则忽略）
- preferredDuringSchedulingIgnoredDuring Execution：优先亲和

```

affinity:
  podAffinity:
    requireDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
            - key: security
              operator: In # 操作符: In, NotIn, Exists, DoesNotExist,Gt,Lt
              values:
                - S1
        topologyKey: kubernetes.io/hostname #设置拓扑域, 默认的还有
        topology.kubernetes.io/region,topology.kubernetes.io/zone

```

2. podAntiAffinity

- requireDuringSchedulingIgnoredDuringExecution: 强制反亲和
- preferredDuringSchedulingIgnoredDuring Execution: 优先反亲和

Node亲和调度

1. NodeSelector: 强制调度到指定节点

2. NodeAffinity

- requireDuringSchedulingIgnoredDuringExecution强制亲和
- preferredDuringSchedulingIgnoredDuring Execution: 优先亲和

3. Taint: 一个node可以有多个taints

1. 行为模式有:

1. NoSchedule: 禁止pod调度来
2. PreferNoSchedule: 尽量不调度来
3. NoExecute: 驱逐没有toleration的pod, 并禁止调度新的

```

apiVersion: v1
kind: Node
metadata:
  name: demo
spec:
  taints:
    - key: "k1"
      value: "v1"
      effect: "NoSchedule"

```

2. Pod的toleration

```

apiVersion: v1
kind: Pod
metadata:
  namespace: demo
  name: demo
spec:

```

```
containers:
- image: busybox
  name: demo
tolerations:
- key: "k1"
  operator: "Equal"
  value: "v1"          #当op=Exist, 可以为空
  effect: "NoSchedule" #可以为空, 匹配所有
  tolerationSeconds: 3600 #taint添加后, pod还能再node上运行时间
```

优先级调度

根据pod配置的priority, 得分高的会对优先级低的进行资源抢占

1. 使用PriorityClass创建优先级, pod中通过priprityClassName引用相应优先级

```
apiVersion: v1
kind: PriorityClass
metadata:
  name: high
value: 1000
globalDefault: false
```

2. 系统默认优先级

1. 没有设置均为0
2. 用户可配置最大优先级为10亿, 系统级别优先级为20亿
3. 内置系统优先级: system-cluster-critical, system-node-critical

pod调度过程

用户提交yaml文件, webhook controller会先对文件进行校验, 校验成功后, api server会创建pod对象, 此时pod的node名称为空。kube scheduler会watch到pod的创建、同时发现node名称为空, 会进行查找合适的node, 并修改pod名称为相应的node名。相应节点上的kubelet watch到pod需要创建, 会在节点上创建容器和网络。

InitContainer

比普通容器先启动, 执行成功后普通才能启动; 且是顺序执行的, 普通容器可以并发启动。

Sidecar 容器模式

用一个辅助容器做辅助工作, 如日志收集、debug、前置操作等。好处: 对业务代码没有侵入; 功能解耦, 共享。模式包括:

- 代理模式: 代理网络等
- 适配器模型: 适配不同外部需求

容器内使用pod信息

1. 通过环境变量

```
env:  
  - name: NODE_NAME  
    valueFrom:  
      fieldRef:  
        fieldPath: spec.nodeName # 根据配置层级用.引用
```

2. 通过配置文件

```
volumes:  
  - name: podinfo  
    downwardAPI: # 使用该API, 支持pod和container的信息  
      items:  
        - path: "labels"  
          fieldRef:  
            fieldPath: metadata.labels
```

Pod健康检查

有三类探针,

- LivenessProbe: 判断是否存活
- ReadinessProbe: 判断是否可用
- StartupProbe: 有且仅有一次的超长启动延时

探针的实现方式:

- ExecAction: 容器内部运行命令, 返回码为0, 代表健康

```
livenessProbe:  
  exec:  
    command:  
      - cat  
      - /tmp/health
```

- TCPSocketAction: 通过容器IP+端口, 执行TCP检查, 能够建立TCP连接, 则健康

```
livenessProbe:  
  tcpSocket:  
    port: 80
```

- HTTPGetAction: HTTP get响应状态码在200~400间, 则认为容器健康

```
livenessProbe:  
  httpGet:  
    path: /health  
    port: 80  
  initialDelaySeconds: 30 # 容器首次健康检查等待时间  
  timeoutSeconds: 1 #等待健康检查响应的时间
```

Labels、Selectors、Annotations、Deployment、Replicaset

Metadata字段概念

Labels

标识型key: value元数据，用于筛选资源，组合资源的唯一方式

Selector

用于类似SQL语句，对Labels进行选择

annotations

用于存储资源的非标识性信息，扩展资源的spec/status。

ownerefence

表明父级资源对象，方便方向查找和级联删除

Deployment

deployment对象控制Replicaset的版本，而replicaset控制集群中pod的数量。

常见操作

- 更新镜像

```
kubectl set image deployment.v1.apps/deployment名称 要更新的Container名=新的镜像名
```

- 回滚

```
kubectl rollout undo deployment deployment的名称
```

- 修改副本数

```
kubectl scale deploy nginx --replicas 5
```

- 查看历史版本

```
kubectl rollout history deploy nginx --revision=N
```

N为第几个版本，没有--revision则查看所有历史记录

Deploy更新策略

在spec.strategy中定义更新策略，两种方式：

- Recreate：所有pod杀掉，再更新
- RollingUpdate：滚动更新pod；spec.strategy.rollingUpdate.maxUnavailable用于指定更新过程中不可用状态的Pod数量的上限；spec.strategy.rollingUpdate.maxSurge用于指定更新过程中pod总数量超过期望数量的部分的最大值

其它类别Pod控制器

Job、cronJob、DaemonSet都是直接管理pod的

1. job

特点是可以并行、串行编排pod运行，设置失败次数。restartPolicy控制失败策略，backoffLimit控制重试次数

2. cronJob

定时执行，schedule定义时间；startingDeadlineSeconds定义job最长运行时间；concurrencyPolicy是否运行并行；

3. DaemonSet

每个节点都运行一个相同的pod

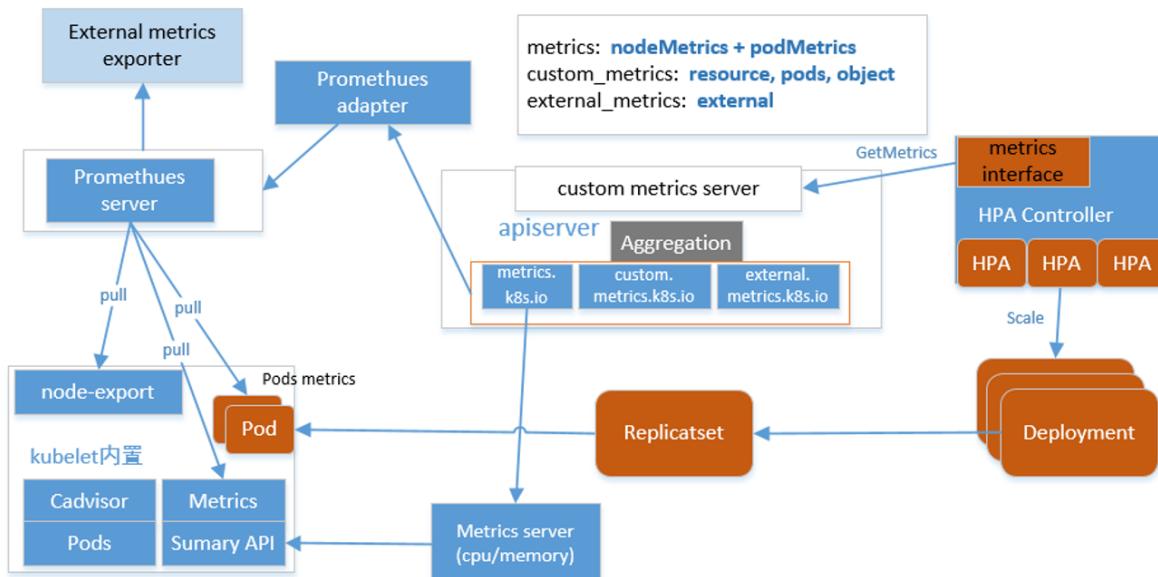
4. StatefulSet

- 每个节点有固定的身份ID，是按数字升序定义，不包含随机字符，成员通过ID相互发现与通信
- 集群规模固定，不能随意变更
- 每个节点都是有状态的，会持久化数据到永久存储中，每个节点重启后都使用之前的数据
- 节点的启动与关闭顺序通常也是有序的

HPA横向自动扩容

通过跟踪分析指定Deployment控制的所有目标Pod的负载变化情况，确定是否需要针对性的调整目标Pod的副本数量。自定义HPA是根据CPU利用率

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    kind: Deployment
    name: php-apache
  targetCPUUtilizationPercentage: 90
```



<https://blog.csdn.net/NetEaseResearch>

VPA垂直扩缩容

- 相对HPA，自动配置Pod的资源请求，降低运维成本（根据pod运行的历史数据和实时资源情况，使用推荐模型，计算pod推荐使用资源值，再更新pod的资源配置或者驱逐pod）

Configmap

好处：配置与容器解耦

限制：命名空间内的cm才能被容器引用；无法用于静态pod；大小限制1MB

定义

```

apiVersion: v1
data: # 该部分是这个cm的value部分
  profile-name: chengzhiyuan
  user: chengzhiyuan@pingan.com.cn
kind: ConfigMap
metadata:
  name: default-install-config-2tk27494b6 #该cm的key

```

使用方法

- 生成容器内环境变量

```

apiVersion: v1
kind: Pod
metadata:

```

```

name: test
sepc:
  containers:
    - name: test
      image: busybox
      env:
        - name: EnvVarName
          valueFrom:
            configMapKeyRef: #关联configmap
              name: cm-name
              key: key-of-cm

    - name: test2
      image: busybox
      envFrom: # 将cm中所有key, value生成环境变量
      - configMapRef
        name: cm-name

```

2. 以volume形式挂载

```

apiVersion: v1
kind: Pod
metadata:
  name: test
sepc:
  containers:
    - name: test
      image: busybox
      volumeMounts:
        - name: volName
          mountPath: /configfiles
  volumes:
    - name: volName
      configMap: #关联configmap
        name: cm-name
        items:
          - key: key-of-cm
            path: configfile

```

secret

定义方式和configMap相同，多了一个type字段，其默认为Opaque，会对value部分进行base64加密。同样，限制文件大小1MB

通过volume形式挂载使用

```

apiVersion: v1
kind: Pod

```

```

metadata:
  name: test
sepc:
  containers:
    - name: test
      image: busybox
      volumeMounts:
        - name: volName
          mountPath: /configfiles
  volumes:
    - name: volName
      secret: #关联secret
      secretName: secret-name

```

#

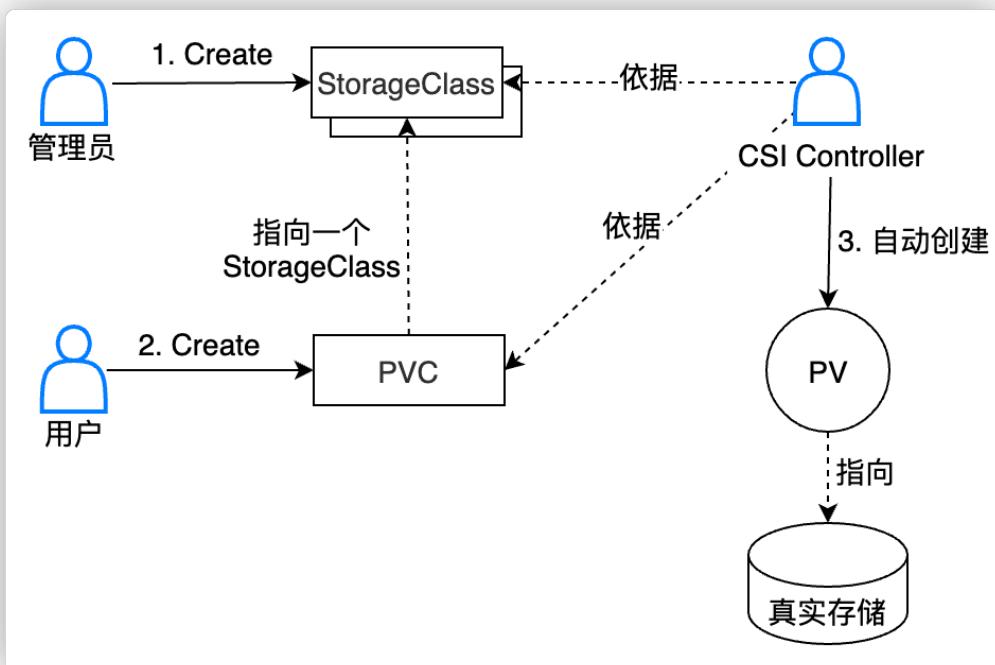
Volume

Volume是pod中由pause容器创建，被pod中所有容器共享，生命周期与pod相同。

- emptyDir: 临时空间
- hostPath: 挂载宿主机上的文件和目录，可以用于持久化数据（会随pod调度变换主机），可以访问宿主机的docker引擎

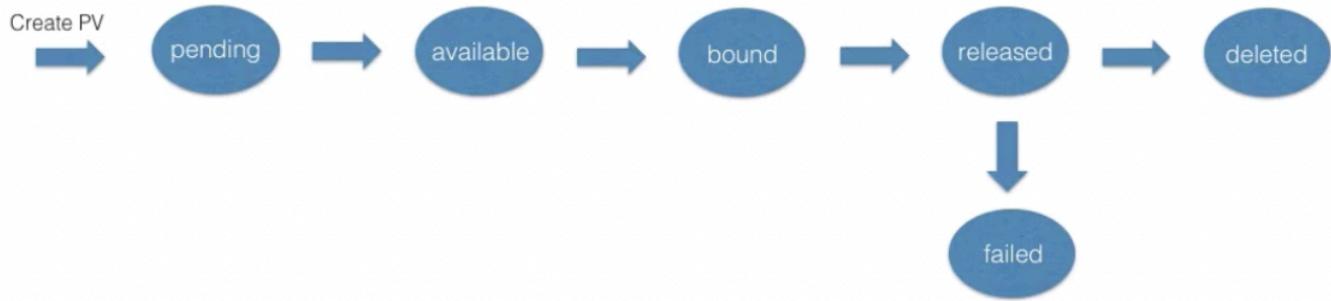
PV PVC StorageClass

k8s中存储是由PV表示的。为了动态根据用户需求创建PV，系统管理员会先定义好创建PV的模版StorageClass，用户在使用时只需要引用对应的模版并写出所用存储的大小、读写权限等即可，这些信息记录在PVC中。存储插件控制器会根据StorageClass和PVC中其他信息自动创建。



PV状态的流转

PV在创建开始后，处于pending，直到创建完成状态变为available。当PVC和PV绑定后，状态变为bound。当绑定的PVC删除，则变为released。released后的PV无法进行再次和PVC绑定。



Volume Plugins

根据源码的位置分为In-Tree和Out-of-Tree两类，与k8s一起发布管理的是In-Tree。独立存在的，由存储实现的是Out-of-Tree，包括CSI/Flexvolume。

存储快照

存储快照的设计其实是仿照 pvc & pv 体系的设计思想。当用户需要存储快照的功能时，可以通过 VolumeSnapshot 对象来声明，并指定相应的 VolumeSnapshotClass 对象，之后由集群中的相关组件动态生成存储快照以及存储快照对应的对象 VolumeSnapshotContent。

PVC 对象将其的 dataSource 字段指定为 VolumeSnapshot 对象，这样就可以恢复数据。

PV

local storage

1. 创建本地PV

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
```

```
path: /data/k8s_pv/volume1
nodeAffinity:
  required:
    nodeSelectorTerms:
      - matchExpressions:
          - key: kubernetes.io/hostname
            operator: In
            values:
              - efa-blpr600005
```

Storage Class

本质是自动化根据PVC创建PV并绑定，配置的参数包括：

- Provisioner存储提供者
- Reclaim Policy：资源回收策略，包括Delete（默认）和Retain
- Allow Volume Expansion：是否运行扩容，需要Volume底层支持该功能
- Volume Binding Mode：绑定模式，Immediate（默认）和WaitForFirstConsumer

1. service作用：负载均衡和服务发现。访问方式：

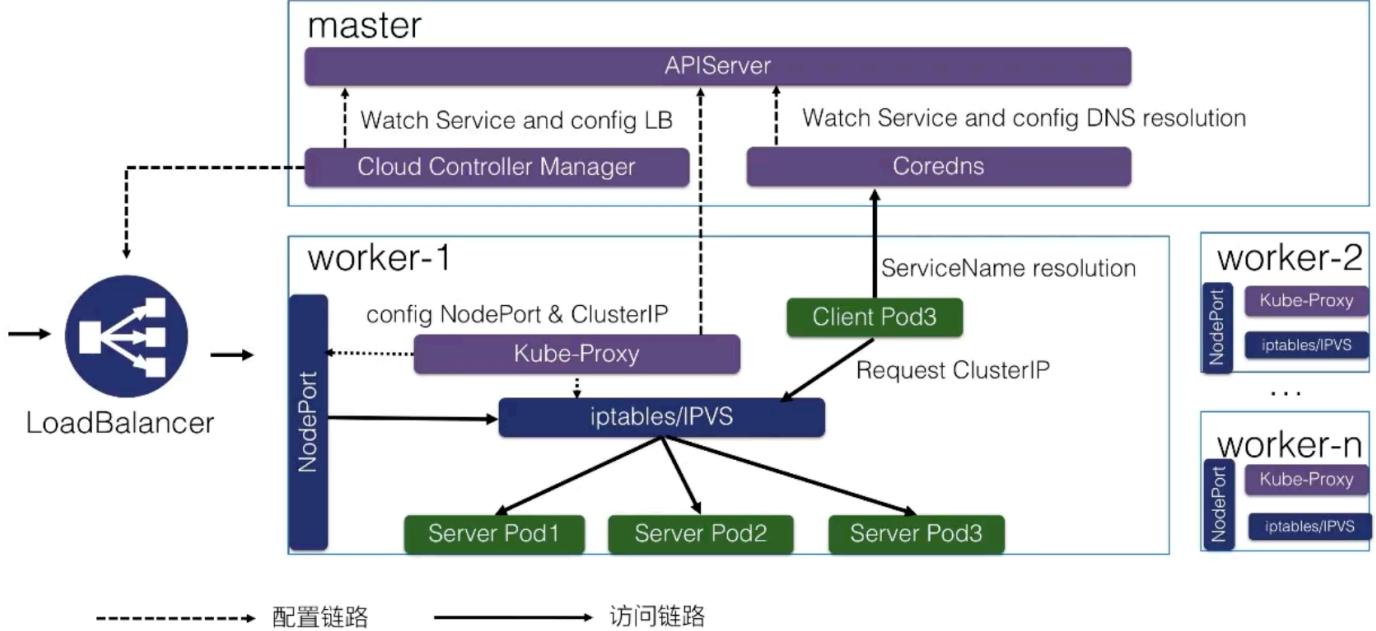
- ClusterIP：默认会生成一个虚拟IP，加上端口，便可以为pod提供访问
- 服务名：依靠k8s DNS解析，通过namespace.serviceName: port即可访问
- 通过环境变量：在同一个 namespace 里的 pod 启动时，K8s 会把 service 的一些 IP 地址、端口，以及一些简单的配置，通过环境变量的方式放到 K8s 的 pod 里面。在 K8s pod 的容器启动之后，通过读取系统的环境变量比读取到 namespace 里面其他 service 配置的一个地址，或者是它的端口号等等。比如在集群的某一个 pod 里面，可以直接通过 curl \$ 取到一个环境变量的值，比如取到 MY_SERVICE_SERVICE_HOST 就是它的一个 IP 地址，MY_SERVICE 就是刚才我们声明的 MY_SERVICE，SERVICE_PORT 就是它的端口号，这样也可以请求到集群里面的 MY_SERVICE 这个 service。

2. Headless service：Pod通过service name直接解析到所有后端Pod IP，客户端自己选择。在定义中设置 clusterIP

3. 向外暴露服务

- NodePort：每个node都会暴露集群中服务的端口，通过每个node都可以路由到指定pod
- LoadBalancer：在NodePort上面加了load balance，把所有它接触到的流量负载均衡到每一个集群节点的 node pod 上面去。然后 node pod 再转化成 ClusterIP，去访问到实际的 pod 上面

4. k8s服务发现原理图



5. service的port

```

apiVersion: v1
kind: Service
metadata:
  labels:
    name: app1
  name: app1
  namespace: default
spec:
  type: NodePort
  ports:
    - port: 8080 #service暴露在cluster ip上的端口, <cluster ip>:port 是提供给集群内部客户访问service的入口
      targetPort: 8080 #targetPort是pod上的端口, 从port和nodePort上到来的数据最终经过kube-proxy流入到后端pod的targetPort上进入容器
      nodePort: 30062 # <nodeIP>:nodePort 是提供给集群外部客户访问service的入口
  selector:
    name: app1

```

6. service原理

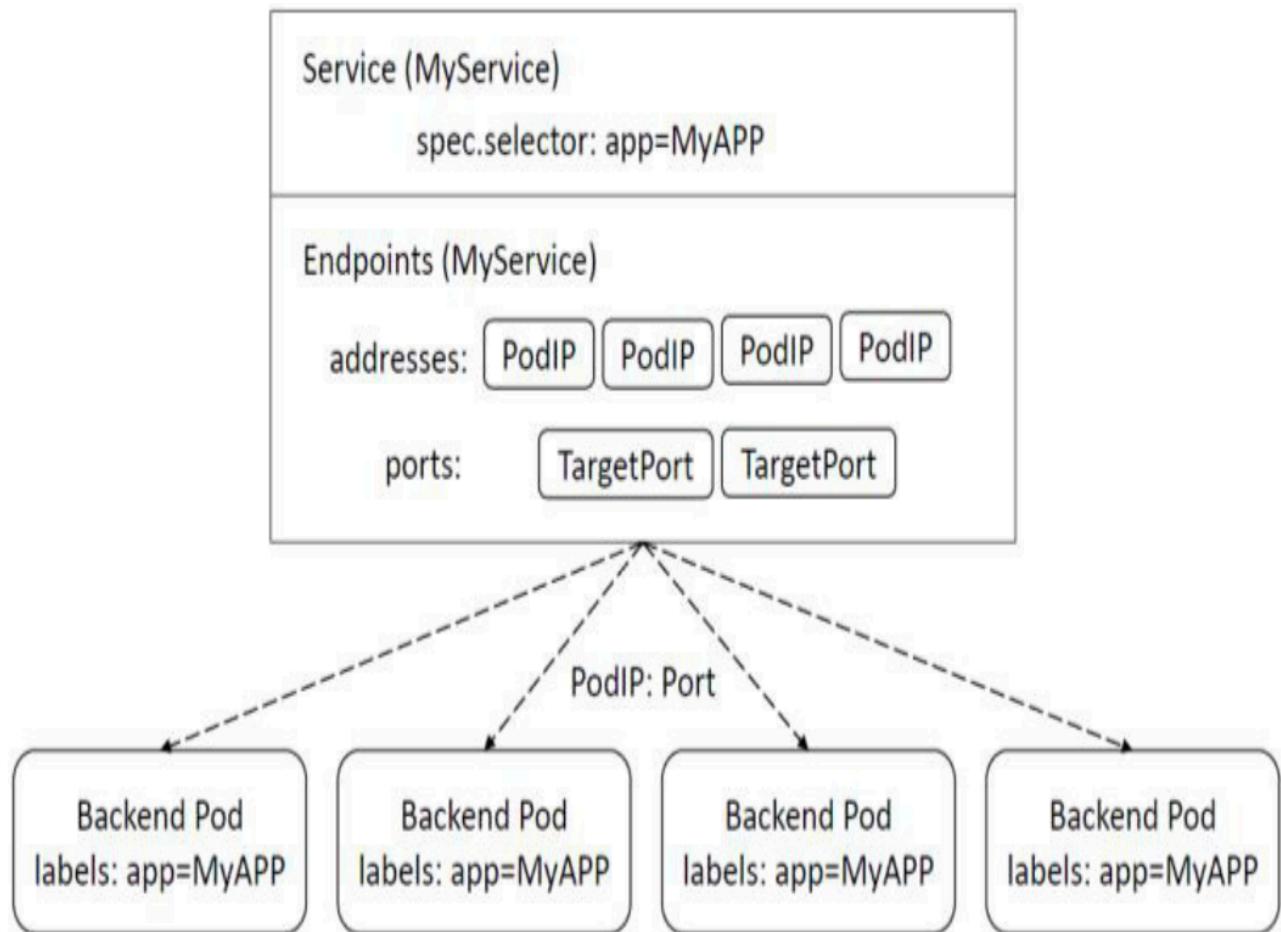
service借助每个Node上的kube-proxy程序实现其功能。service相当于一组Pod的LB， kube-proxy管理service的访问入口和service的Endpoints。

kube-proxy代理模式有四种：

- userspace：由kube-proxy实现，效率最低
- iptables：通过设置iptables实现service到endpoint的分发，效率高；但某个endpoint不可用，会导致相应失败
- ipvs：通过Linux Kernel的netlink接口设置IPVS规则，效率和吞吐率最高；需要系统开启IPVS模块；且支持较多的负载均衡策略
- kernelspace：windows server上用的

7. Service, Endpoint, pod关系

Service会指向endpoint, endpoint记录对应的pod的ip



8. 常用命令

- 会话保持机制：保证请求转发到相同pod，配置service.spec.sessionAffinity = ClientIP
- 将外部服务定义为service
 - 创建service时不需要设置Label Selector
 - 定义一个与service关联到Endpoint资源

```
apiVersion: v1
kind: Endpoint
metadata:
  name: my-service
subsets:
- addresses:
  - IP: 1.2.3.4
  ports:
  - port: 80
```

- 快速创建service

```
kubectl expose deploy webapp
```

概念

overlay underlay

容器网络寄生在主机网络之上，容器网络实现方案有两种类型：

- Underlay: 与Host网络同层
- Overlay: 不需要Host网络的IPM申请IP，只要和Host不冲突即可

二层网络 三层路由

- 二层网络：是通过MAC地址获取目标，在数据链路层进行数据传递；好处是不需要解封包，效率高；坏处是所有机器必须同在一个子网，只能通过交换机连接，机器数量有限
- 三层路由：需要通过IP进行寻址；好处是网络可以更加复杂、规模更大；坏处是性能相对二层网络低

容器网络模型

CNM与CNI网络模型

- CNM通过Network sandbox、Endpoint、Network实现。容器中会有一个Network sandbox，network sandbox中会有一个或多个Endpoint；Network实现多个Endpoint的相连，从而实现容器的网络连通
- CNI表示通过绑定网络插件的模式加入网络中，它只定义了容器运行环境与网络插件之间的接口规范，网络插件则由不同的供应商实现；容器拥有独立的Linux网络命名空间，可以绑定多个网络插件加入多个网络；

k8s采用的CNI

k8s采用CNI的网络模型，本身不自带网络控制，由第三方组件进行网络实现。网络模型的要求是IP-per-Pod：

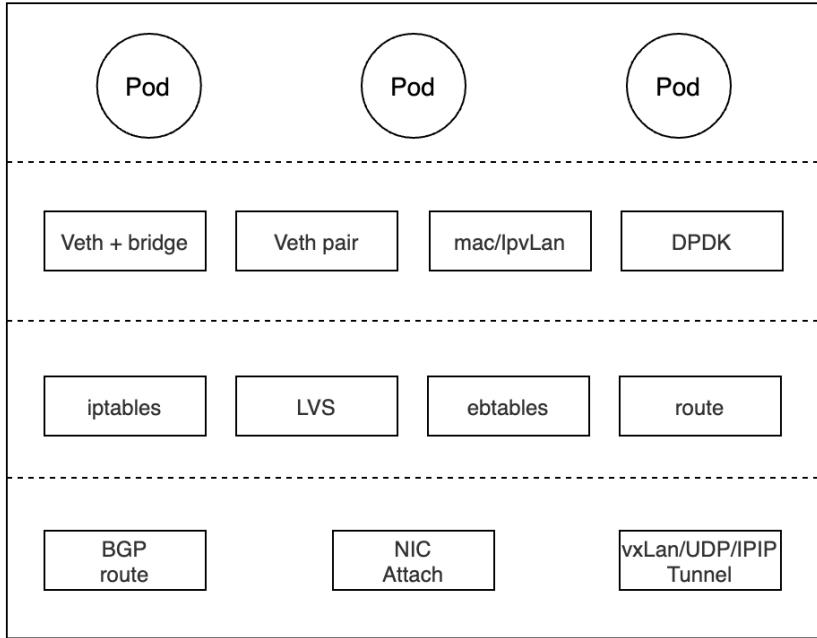
- 第一条：任意两个 pod 之间其实是可以直接通信的，无需经过显式地使用 NAT 来接收数据和地址的转换；
- 第二条：node 与 pod 之间是可以直接通信的，无需使用明显的地址转换；
- 第三条：pod 看到自己的 IP 跟别人看见它所用的IP是一样的，中间不能经过转换。

网络模型的目标：

- 外部世界和 service 之间是怎么通信的？
- service 如何与它后端的 pod 通讯？
- pod 和 pod 之间调用是怎么做到通信的？
- 最后就是 pod 内部容器与容器之间的通信？

容器网络方案由三部分构成：接入+流控+通道

- 接入：是容器与宿主机用哪种机制做连接，如：Veth+bridge，Veth+pair，Mac，IPvlan
- 流控：看方案是否支持network policy，以及怎么支持
- 通道：两个主机通过什么方式完成包的传输



接入：桥接、ipvlan派生等不同方式，否则容器数据包进出容器，作为单独的CNI功能插件

流控：支持network policy功能落地，可采用Felix-iptables, ebttables, eBPF-XDP方式或者Hook在数据路径上（Data Path），可作为单独add-on或CNI插件方式部署

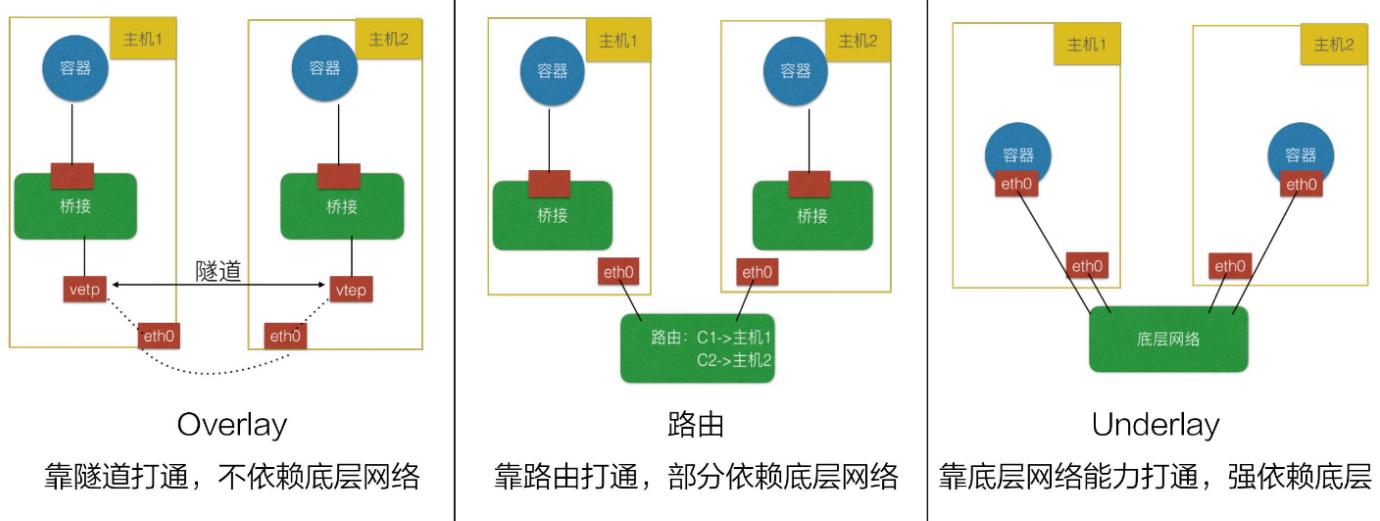
通道：有BGP路由（Felix BGP agent），直接挂接外网卡，各种隧道，纯Gateway路由等方式，负责节点间通讯，作为单独功能CNI插件

主流容器网络实现方案

CNI三种实现模式：

哪个CNI插件适合我

CNI插件通常有三种实现模式



阿里云

CLOUD NATIVE COMPUTING FOUNDATION

Flannel

它提供了多种的网络 backend。不同的 backend 实现了不同的拓扑，它可以覆盖多种场景

Calico

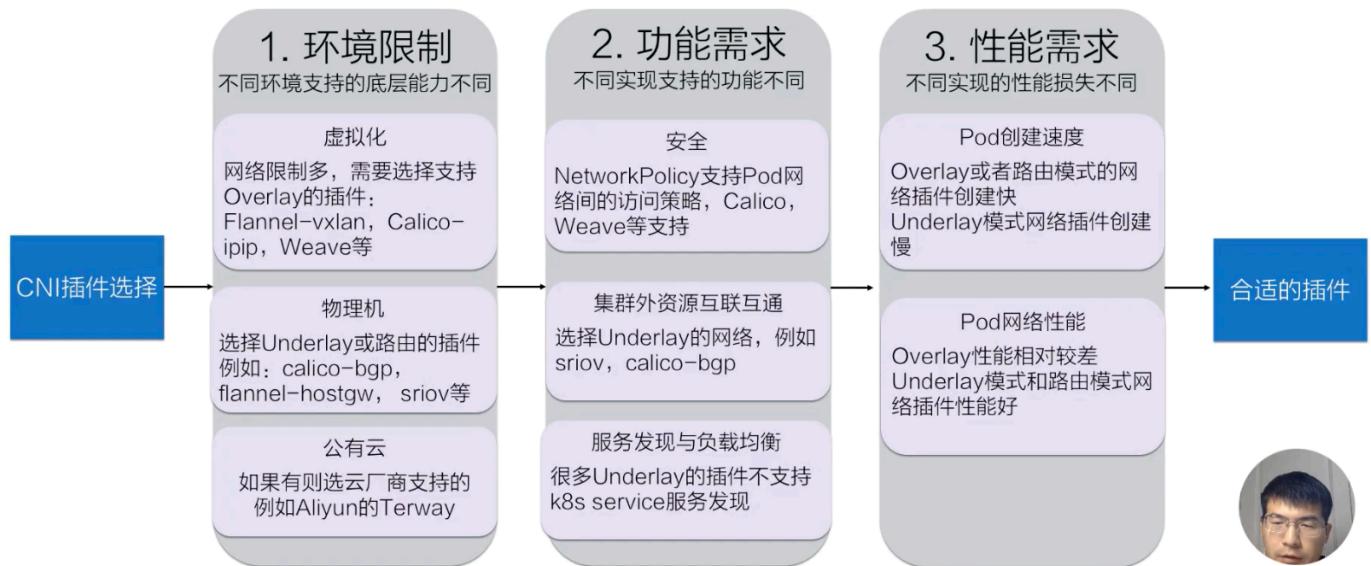
主要是采用了策略路由，节点之间采用 BGP 的协议，去进行路由的同步。它的特点是功能比较丰富，尤其是对 Network Point 支持比较好，大家都知道 Calico 对底层网络的要求，一般是需要 mac 地址能够直通，不能跨二层域；

Canal

WeaveNet

对数据做一些加密

选择CNI的方法



Network Policy

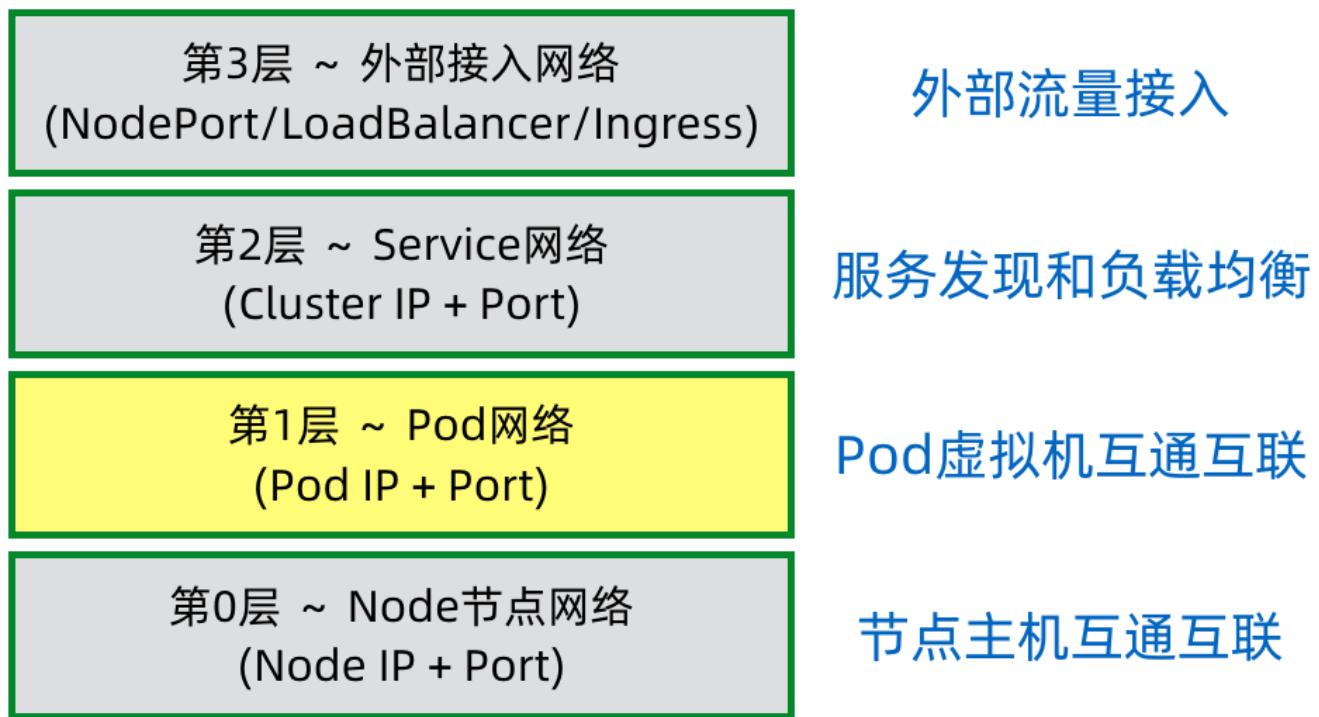
用于网络隔离，相当于一个pod网路连通的白名单。基本原理是：通过选择器找到一组pod，再通过流的特征描述决定他们之间能否连通。选用的CNI插件要能够支持Network Policy，k8s没有内在实现。

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: backend-policy
  namespace: development
spec:
  podSelector: #控制对象选择器
  matchLabels:
    app: webapp
    role: backend
  ingress: # 流入控制
  - from:
    - namespaceSelector: {}
      podSelector:
```

```
matchLabels:  
  app: webapp  
  role: frontend  
engross: #流出控制  
- to:  
  - ipBlock:  
    cidr: 10.0.0.0/24
```

Pod网络

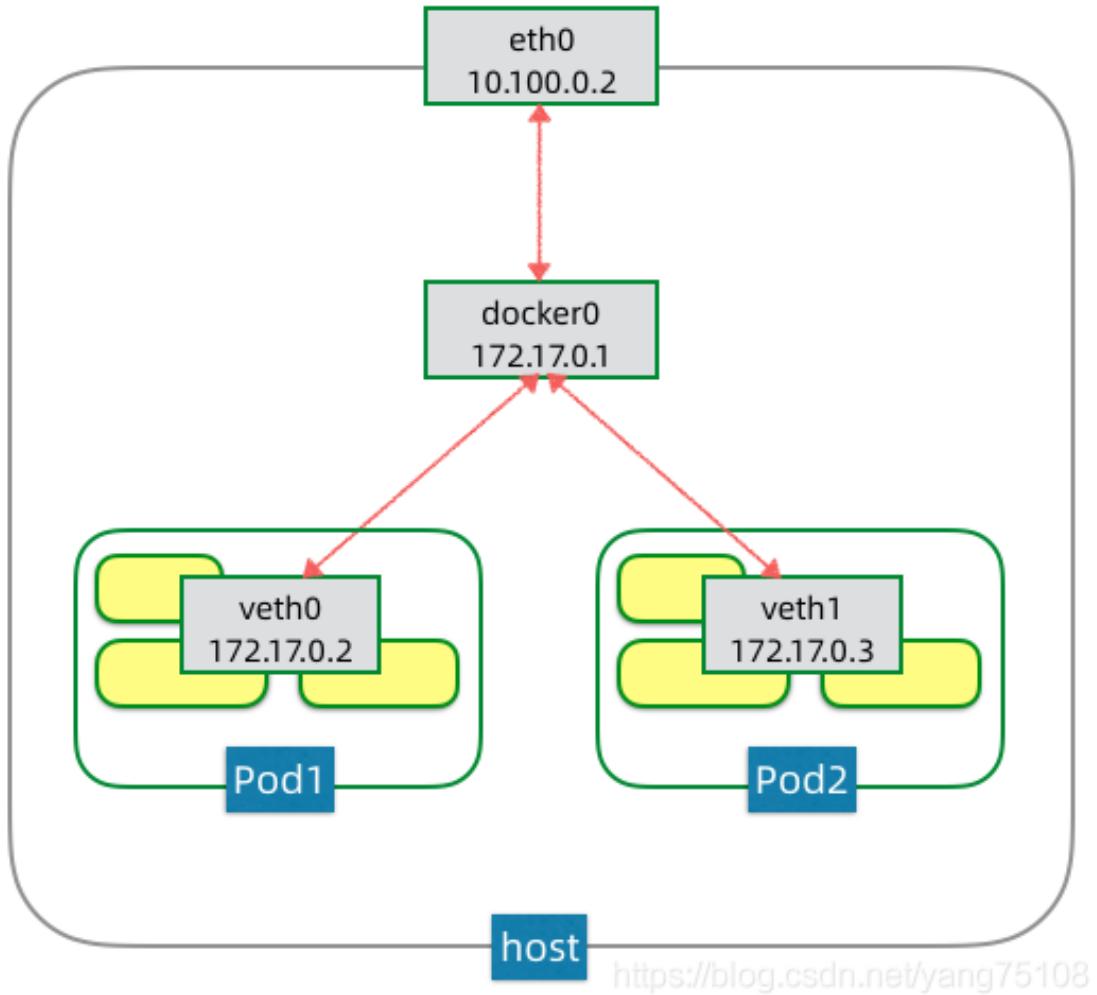
四层网络



<https://blog.csdn.net/yang75108>

Pod网络依赖

同一节点的Pod网络



<https://blog.csdn.net/yang75108>

eth0节点主机上的网卡：支持该节点流量出入，互通的设备

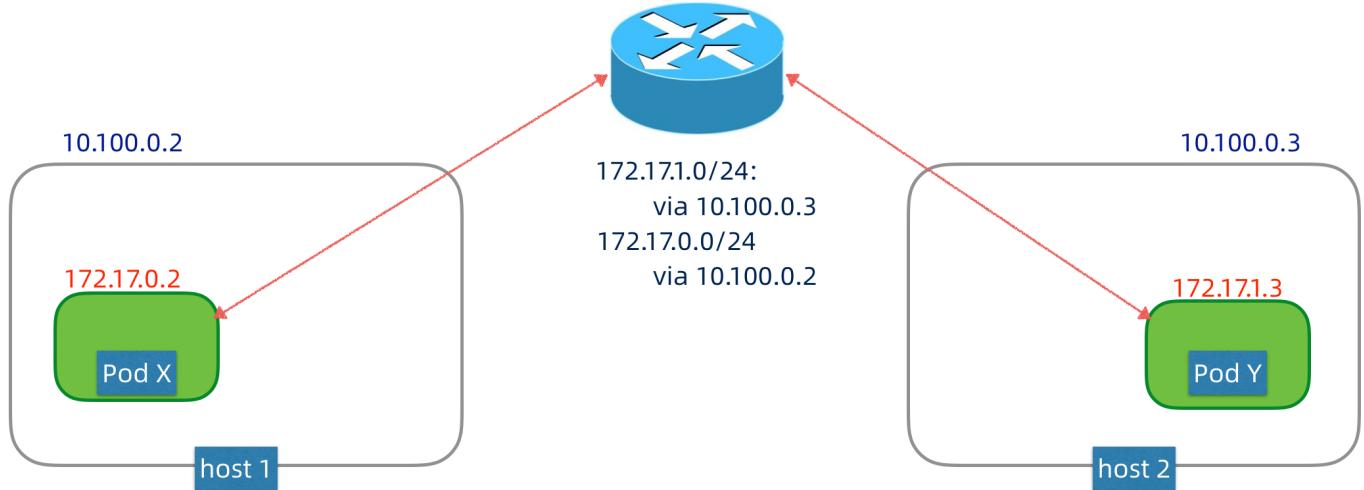
docker0：虚拟网桥，是虚拟交换机，支持该节点上的Pod间进行IP寻址和互通的设备

veth0:是Pod的虚拟网卡，支持Pod内容器互通和对外访问的虚拟设备。其由pause容器建立

Pod的IP由docker0分配

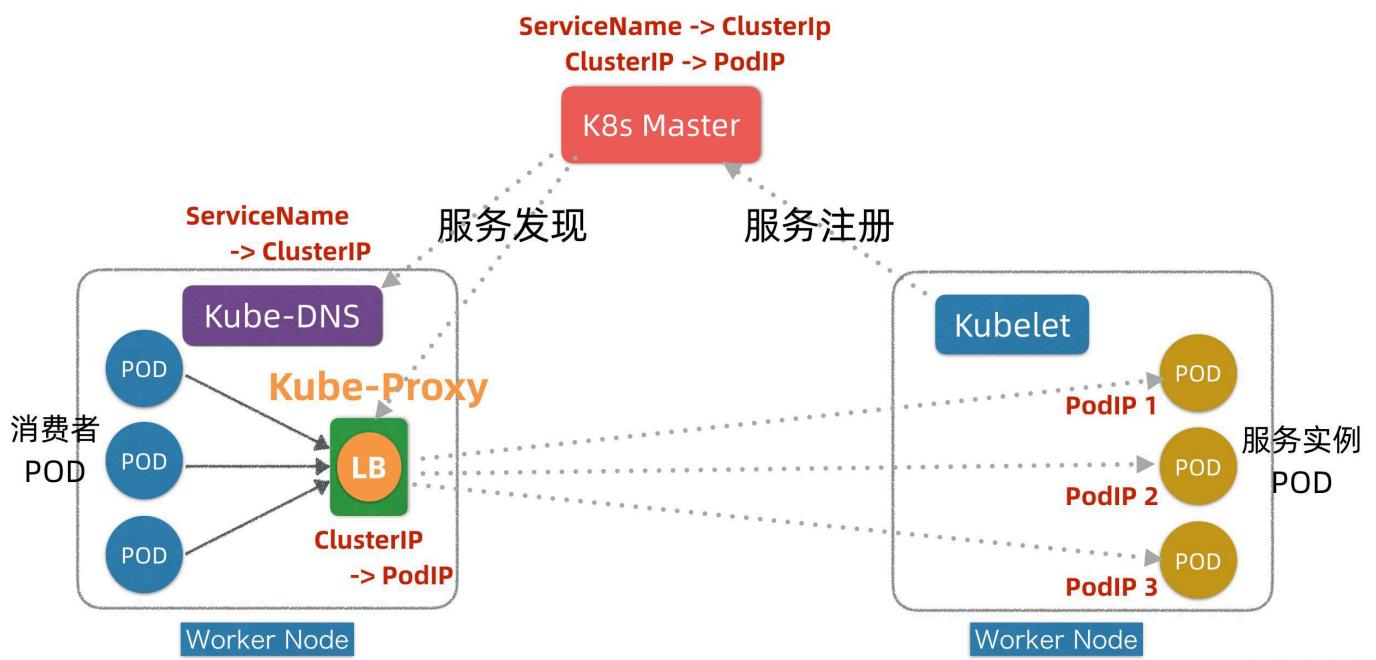
不同节点的Pod网络

- 方案一路由方案：用路由设备为K8s集群的Pod网络单独划分网段，并配置路由器支持Pod网络转发

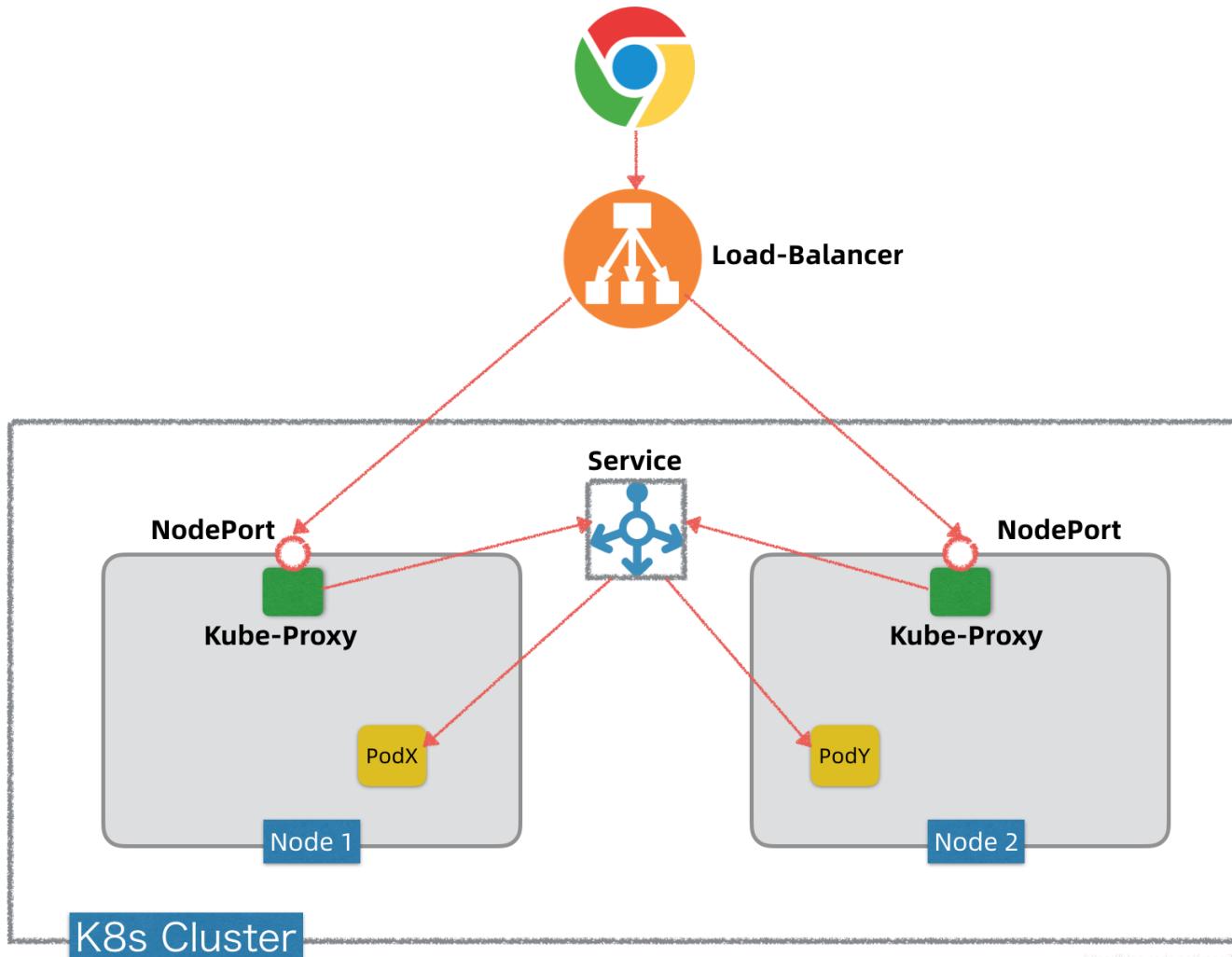


2. 方案二覆盖网络：现有网络基础上建立虚拟网络；Pod网络数据包出节点前封装为节点网络的数据包，达到目标节点再解封出来，转发给内部Pod网络。有额外封包解包性能开销

Service网络



对外暴露网络



<https://blog.csdn.net/yang75108>

NodePort是K8s内部服务对外暴露的基础，LoadBalancer底层有赖于NodePort。NodePort背后是Kube-Proxy，Kube-Proxy是沟通Service网络、Pod网络和节点网络的桥梁。

将K8s服务通过NodePort对外暴露是以集群方式暴露的，每个节点上都会暴露相应的NodePort，通过LoadBalancer可以实现负载均衡访问。公有云(如阿里云/AWS/GCP)提供的K8s，都支持自动部署LB，且提供公网可访问IP，LB背后对接NodePort。

Ingress扮演的角色是对K8s内部服务进行集中反向代理，通过Ingress，我们可以同时对外暴露K8s内部的多个服务，但是只需要购买1个(或者少量)LB。Ingress本质也是一种K8s的特殊Service，它也通过HostPort(80/443)对外暴露。

通过Kubectl Proxy或者Port Forward，可以在本地环境快速调试访问K8s中的服务或Pod。

K8s的Service发布主要有3种type，type=ClusterIP，表示仅内部可访问服务，type=NodePort，表示通过NodePort对外暴露服务，type=LoadBalancer，表示通过LoadBalancer对外暴露服务(底层对接NodePort，一般公有云才支持)。

	作用	实现
节点网络	Master/Worker 节点之间网络互通	路由器, 交换机, 网卡
Pod 网络	Pod 虚拟机之间互通	虚拟网卡, 虚拟网桥, 网卡, 路由器 or 覆盖网络
Service 网络	服务发现+负载均衡	Kube-proxy, Kubelet, Master, Kube-DNS
NodePort	将 Service 暴露在节点网络上	Kube-proxy
LoadBalancer	将Service暴露在公网上+负载均衡	公有云 LB + NodePort
Ingress	反向路由, 安全, 日志监控 (类似反向代理 or 网关)	Nginx/Envoy/Traefik/Zuul/SpringCloudGateway...

<https://blog.csdn.net/yang75108>

Ingress

ingress提供根据路由规则直接将客户请求转发到Endpoint上，跳过kube-proxy设置的转发规则，提高效率。
Ingress由Ingress Controller和Ingress对象组成，Ingress对象实质是controller的路由规则配置，controller根据ingress配置实现真正路由转发；controller有deployment管理。

Readiness指针判断服务是否就绪，只有就绪流量才会接入

liveness指针判断服务是否正常，不健康则杀Pod

探测方式：

- httpGet: 返回200-399则认为健康
- Exec: 执行命令，命令返回值为0则认为健康。最好用编译的程序来判断，性能会比shell命令提高30%
- tcpSocket: 通过IP和port的tcp能够正常建立，则认为健康

常用配置：

- initialDelaySeconds: 启动后延迟检查
- periodSeconds: 探测周期
- timeoutSecond: 探测超时

故障排查

1. Pod停留在Pending

调度器没有介入，通过kubectl describe pod查看实践，通常与资源使用相关

2. Pod停留在waiting

通常是镜像拉取有问题，看看镜像网路或镜像地址是否正确

3. Pod不断被拉起且看到crashing

pod完成调度并启动，但启动失败。通常是配置、权限造成，需要查看pod日志

4. Pod处于running但工作不正常

通常是部分字段拼写错误造成但，校验部署排查，如：kubectl apply --validate -f pod.yaml

5. service无法正常工作

在排查网络插件问题后，可能是label配置问题，可以查看endpoint的方式检查

远程调试

- 自动补全

```
yum install -y bash-completion
source /usr/share/bash-completion/bash_completion
source <(kubectl completion bash)
echo "source <(kubectl completion bash)" >> ~/.bashrc
```

- 当集群中应用依赖的应用需要本地调试时：

使用Telepresence将本地应用代理到集群中的一个service上

```
Telepresence --swap-deployment $DEPLOYMENT_NAME
```

- 当本地开发的应用需要调用集群中的服务时：

使用port-forward将远程应用代理到本地端口上

```
kubectl port-forward svc/app -n app-namespace
```

- 进入pod中调试

通过kubectl 的一个插件kubectl-debug，能够用工具pod连接目标pod，从而对其进行诊断。

```
kubectl debug 目标pod名
```

监控

k8s监控有三种接口：

- Resource Metrics

对应的接口是 metrics.k8s.io，主要的实现就是 metrics-server，它提供的是资源的监控，比较常见的是节点级别、pod 级别、namespace 级别、class 级别。这类的监控指标都可以通过 metrics.k8s.io 这个接口获取到。

- Custom Metrics

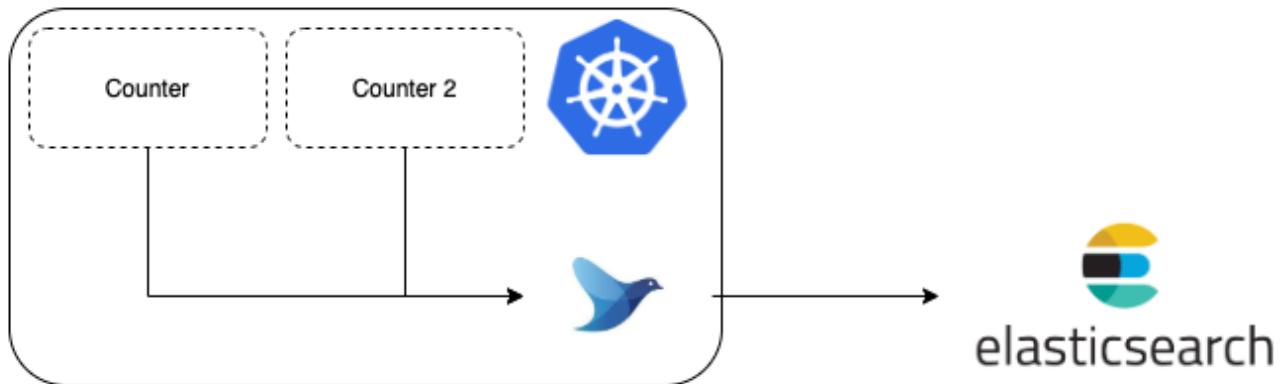
对应的 API 是 custom.metrics.k8s.io，主要的实现是 Prometheus。它提供的是资源监控和自定义监控，资源监控和上面的资源监控其实是有覆盖关系的，而这个自定义监控指的是：比如应用上面想暴露一个类似像在线人数，或者说调用后面的这个数据库的 MySQL 的慢查询。这些其实都是可以在应用层做自己的定义的，然后并通过标准的 Prometheus 的 client，暴露出相应的 metrics，然后再被 Prometheus 进行采集。

- External Metrics

external.metrics.k8s.io。主要的实现厂商就是各个云厂商的 provider，通过这个 provider 可以获取云资源的监控指标。

日志

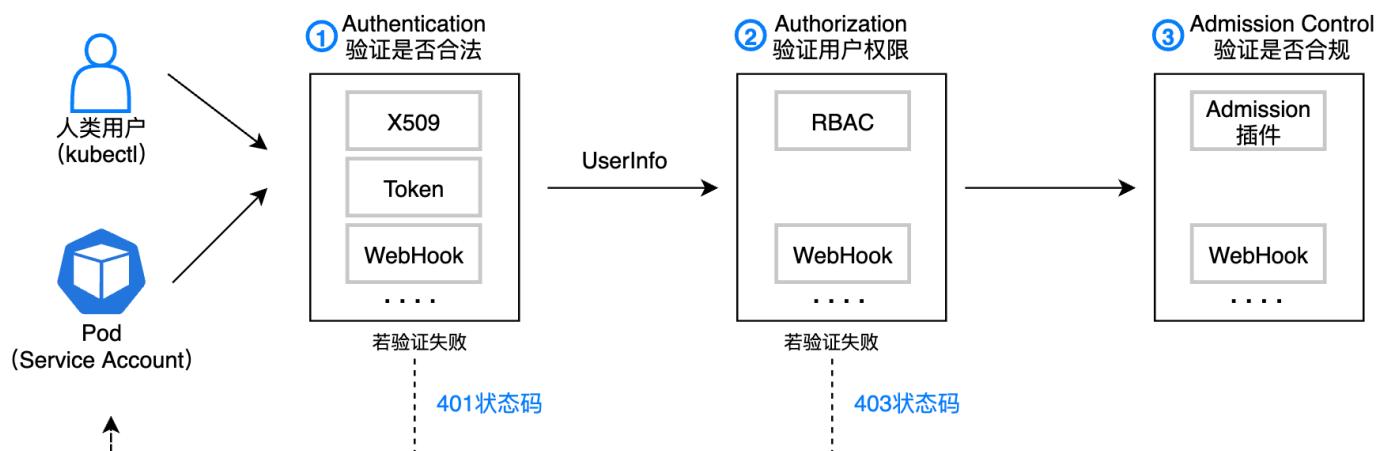
fluentd以sidecar模式收集日志存储到elasticsearch，再在Kibana中展示



k8s集群安全控制包括API访问控制、Pod安全策略、网络安全策略。

API访问控制

k8s有两类用户:人类用户和service account。service account是集群内的进程程序使用的。k8s没有User对象维护人类用户信息，其通过CA认证验证用户的身份。Service account被绑定到特定的命名空间，通过API server创建，保存在secrets中并挂载到pod中。人类用户或者Pod中的进程可能会访问API server，访问过程中的控制分为三个阶段。



Authentication

Authentication的认证有多种方式，可以组合使用。组合情况下，有验证结果的作为该环节的结果。service account会使用token方式验证，普通用户则需要另外再选择一种方式。验证通过的情况下，会得到用户的信息。用户信息中username被后续环节使用。

- X509：客户端和服务端从CA机构获取证书，验证对方证书的有效性，从而确认对方身份。
- HTTP Bearer Token：API server用csv文件保存用户信息，验证HTTP header中的token，获取用户名，并与csv文件对照判断用户是否合法。ServiceAccount本质使用的就是该方法。
- OpenID第三方认证
- Webhook Token：API server从需要认证的请求Header中获取token，发送到Webhook服务进行认证，根据

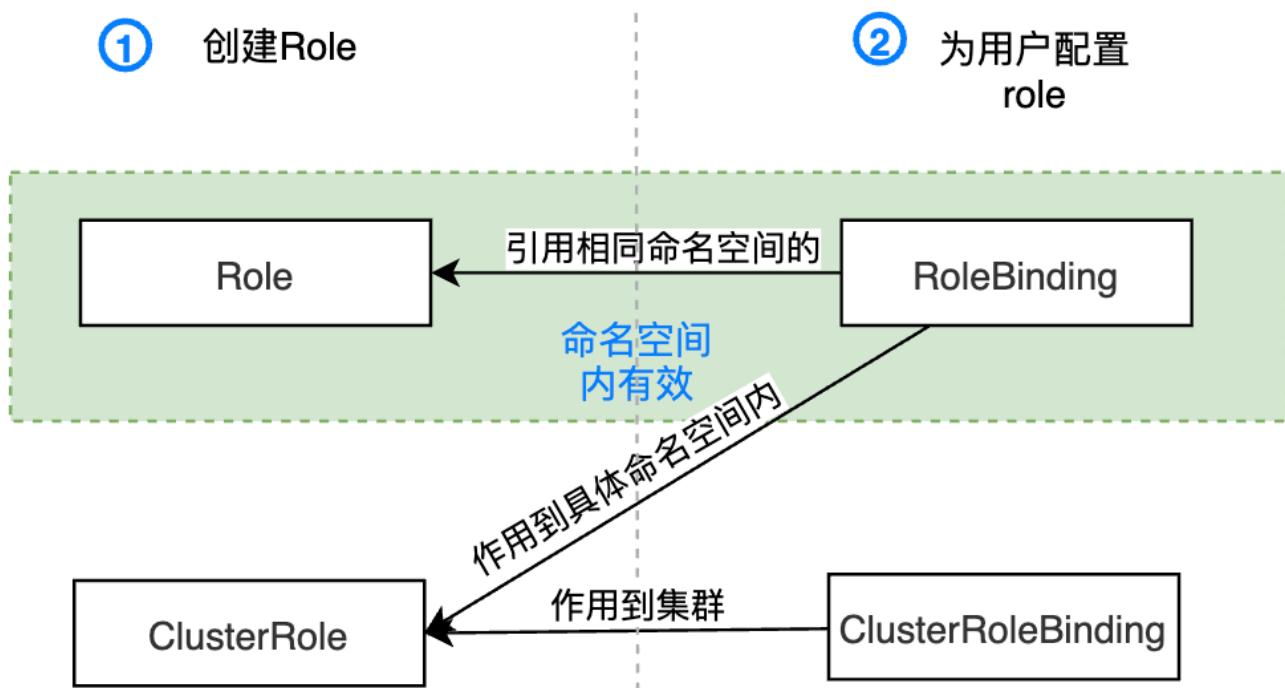
- 返回到结果决定认证有效性。
- Authenticating Proxy (认证代理)

Authorization

授权策略包括：

- AlwaysAllow: 允许所有
- AlwaysDeny: 拒绝所有
- Node: 针对kubelet发起的请求的特殊授权模式，包括读取、写入、授权相关操作的限制
- Webhook: 通过webhook模式获取授权结果
- RBAC: 基于角色进行授权

主要分为两个步骤为用户设置相关权限，首先创建包含一组权限集合的Role/ClusterRole，再创建RoleBinding将权限配置到具体用户身上。Role和RoleBinding只能作用在具体命名空间内。通常会创建若干RoleBinding，再根据需要在各个命名空间引用ClusterRole，从而作用在具体命名空间中。用户无法修改与之绑定的Role或ClusterRole，需要变更时应先删除再从新绑定。



Role、ClusterRole定义

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [ "" ] # "" indicates the core API group
  resources: [ "pods" ]
  verbs: [ "get", "watch", "list" ]
---

```

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [ "" ]
  #
  # at the HTTP level, the name of the resource for accessing Secret
  # objects is "secrets"
  resources: [ "secrets" ]
  verbs: [ "get", "watch", "list" ]

```

RoleBinding、ClusterRoleBinding、ClusterRoleBinding的聚合定义

```

apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "jane" to read pods in the "default" namespace.
# You need to already have a Role named "pod-reader" in that namespace.
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
# You can specify more than one "subject"
- kind: User
  name: jane # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  # "roleRef" specifies the binding to a Role / ClusterRole
  kind: Role #this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or ClusterRole you wish
to bind to
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: rbac.authorization.k8s.io/v1
# This cluster role binding allows anyone in the "manager" group to read secrets in
any namespace.
kind: ClusterRoleBinding
metadata:
  name: read-secrets-global
subjects:
- kind: Group
  name: manager # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
---

```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: monitoring
aggregationRule: #多个roleBinding的聚合
  clusterRoleSelectors:
  - matchLabels:
    rbac.example.com/aggregate-to-monitoring: "true"
rules: [] # The control plane automatically fills in the rules
```

系统默认设置的Role和用户

Pod安全策略

Pod的安全策略是通过设置security context限制不可信的容器行为，从而保护系统和其它容器不受影响。Security context的设置有三种方式：

- 容器级别配置：只影响该容器本身，且会覆盖pod级别的设置，不影响Volume
- pod级别配置：影响pod中的容器及Volume
- Pod Security Policies (PSP)：应用到集群内所有Pod和Volume

pod和容器中的security context配置

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  securityContext:
    privileged: false    #会被container定义的覆盖
    fsGroup: 1234
    supplementalGroups: [5678]
    seLinuxOptions:
      level: "s0:c123,c456"
  containers:
  - name: hello-world-container
    # The container definition
    # ...
    securityContext:
      privileged: true
```

Pod Security Policy

PSP是通过admission插件定义的创建合规性检查

1. 启动PSP准入控制器

kube-apiserver启动时，参数设置为： `--enable-admission-plugins=..., PodSecurityPolicy`

2. 创建一个PSP对象

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: no-privilege
spec:
  privileged: false
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
    - '*'
```

3. 借助RBAC设置PSP

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: no-privilege:no-privilege
rules:
- apiGroups:
- extensions
resources:
- podsecuritypolicies
resourceNames:
- psp
verbs:
- use
---

kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: no-privilege:no-privilege
subjects:
- kind: Group
  name: system:authenticated
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: no-privilege:no-privilege
```

apiGroup: rbac.authorization.k8s.io

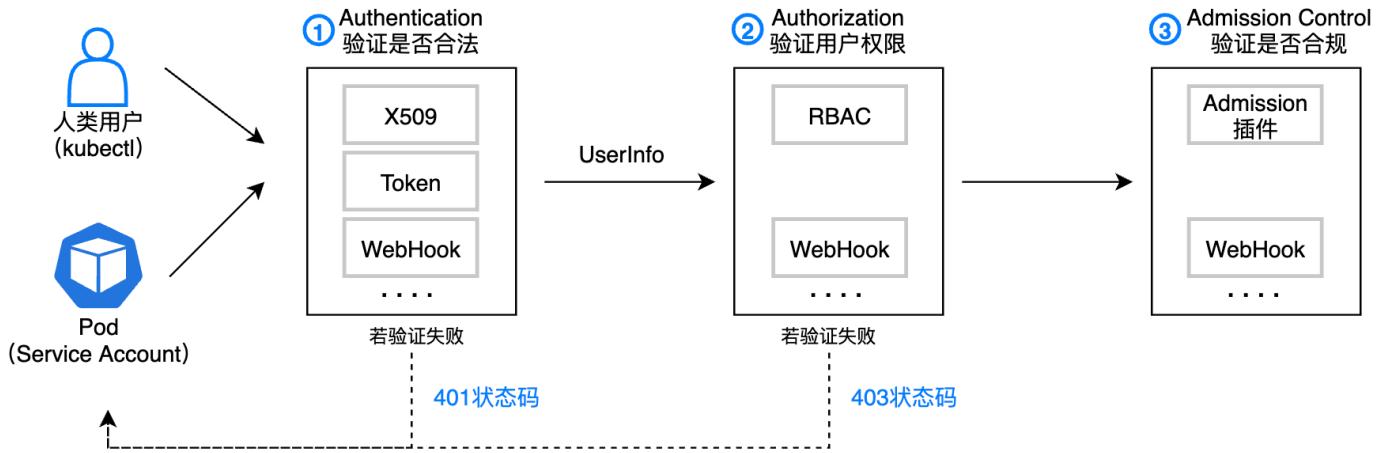
支持的安全策略

分类	控制项	说明
	privileged	运行特权容器
Linux能力相关	defaultAddCapabilities	可添加到容器的 Capabilities
Linux能力相关	requiredDropCapabilities	会从容器中删除的 Capabilities
Linux能力相关	allowedCapabilities	允许使用的 Capabilities 列表
宿主机命名空间相关	hostNetwork	允许使用 host 网络
宿主机命名空间相关	hostPorts	允许的 host 端口列表
宿主机命名空间相关	hostPID	使用 host PID namespace
宿主机命名空间相关	hostIPC	使用 host IPC namespace
	seLinux	SELinux Context
用户和组相关	runAsUser	运行运行容器的user ID范围
用户和组相关	runAsGroup	运行运行容器的Group ID范围
用户和组相关	supplementalGroups	允许的补充用户组
存储卷和文件系统相关	fsGroup	volume FSGroup
存储卷和文件系统相关	volumes	控制容器可以使用哪些 volume
存储卷和文件系统相关	readOnlyRootFilesystem	只读根文件系统
存储卷和文件系统相关	allowedHostPaths	允许 hostPath 插件使用的路径列表
	allowedFlexVolumes	允许使用的 flexVolume 插件列表
提升权限相关	allowPrivilegeEscalation	允许容器进程设置 no_new_privs
提升权限相关	defaultAllowPrivilegeEscalation	默认是否允许特权升级

[官方列表](#)

网络安全策略

作用



创建对象时，校验用户身份后，会通过admission controller的验证。在这一步可以做功能的扩展。

可以定义两种Admission webhook：

- ValidatingAdmissionWebhook
用于验证创建对象，是否允许资源创建
- MutatingAdmissionWebhook
在开始创建对象时，请求会先发到编写的controller中，做一些操作。如：注入操作、优化

开发自己的Admission webhook

前提

- v1.16(to use `admissionregistration.k8s.io/v1`)或者v1.9(to use `admissionregistration.k8s.io/v1beta1`)
- 开启 `--enable-admission-plugins`
- 开启`admissionregistration.k8s.io/v1`或`admissionregistration.k8s.io/v1beta1`

1. 开发webhook server

<https://github.com/kubernetes/kubernetes/blob/release-1.21/test/images/agnhost/webhook/main.go>

2. 部署

需要用deployment和service部署

3. 配置webhook

```

apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: "pod-policy.example.com"
webhooks:
- name: "pod-policy.example.com"
  rules:
    
```

```

- apiGroups:      [ "" ]
  apiVersions:   [ "v1" ]
  operations:    [ "CREATE" ]
  resources:     [ "pods" ]
  scope:         "Namespaced"
clientConfig:
  service:
    namespace: "example-namespace"
    name: "example-service"
    caBundle: "Ci0tLS0tQk...<`caBundle` is a PEM encoded CA bundle which will be used
to validate the webhook's server certificate.>...TLS0K"
  admissionReviewVersions: [ "v1" , "v1beta1" ]
  sideEffects: None
  timeoutSeconds: 5
``1. Node隔离方式:
```

- * 编辑node，设置spec.unschedulable: true
- * 执行: kubectl patch node node-1 -p '{"spec":{"unschedulable":true}}'
- * Kubectl cordon node-1

2. label操作

- * 删除: label名字后加-, 如: kubectl label po pod1 role-
- * 修改: kubectl label po pod1 role=master --overwrite

3. 集群环境隔离

- * 通过定义不同namespace隔离资源
- * 定义Context，可以实现集群切换或者命名空间限制访问

```

```yaml
#不同集群
apiVersion: v1
clusters:
- cluster:
 certificate-authority-data: DATA+OMITTED
 server: https://kubernetes.docker.internal:6443
 name: docker-desktop
- cluster:
 certificate-authority-data: DATA+OMITTED
 server: https://192.168.31.147:6443
 name: kubernetes
contexts:
- context:
 cluster: docker-desktop #约束使用不同的集群
 user: docker-desktop
 name: docker-desktop
- context:
```

```

cluster: kubernetes
user: kubernetes-admin
name: kubernetes-admin@kubernetes
current-context: docker-desktop
kind: Config
preferences: {}
users:
- name: docker-desktop
 user:
 client-certificate-data: REDACTED
 client-key-data: REDACTED
- name: kubernetes-admin
 user:
 client-certificate-data: REDACTED
 client-key-data: REDACTED

```

```

```yaml
#命名空间控制
apiVersion: v1
clusters:
- cluster:
    server: https://kubernetes.docker.internal:6443
    name: docker-desktop
contexts:
- context:
    cluster: docker-desktop
    namespace: development #约束使用命名空间
    user: dev
    name: ctx-dev
- context:
    cluster: kubernetes
    namespace: production
    user: prod
    name: ctx-prod
current-context: docker-desktop
kind: Config
preferences: {}
users: null
```

```

切换context: `kubectl config use-context ctx-dev`

#### 4. 资源限制

- limit, request
- limitRange可以进行全局的最大、最小、默认、比例等限制
- Pod可以定义spec.priorityClassName: high/medium/low划分不同优先级；通过spec.scopeSelector, resource quota可以对不同优先级进行定义

## 5. 拓扑管理器

对硬件配合要求较高对场景，拓扑管理器帮助提供多种资源需求组合，如从相同NUMA节点分配cpu、内存。需要与QoS搭配使用

## 6. Pod驱逐

- nodefs是kubelet用于存储卷系统、服务程序日志等的文件系统；imagefs是容器运行时使用的可选文件系统，用于保存容器镜像和容器可写层数据
- 默认情况下，驱逐条件
  - nodefs.available < 10%
  - nodefs.inodesFree < 5%
  - imagefs.available < 15%
- Kubernetes1.9后，kubelet只根据Pod的nodefs使用量排序，并选择使用量最多的Pod进行驱逐，即是QoS为Guaranteed的Pod也可能被驱逐
- 当节点为MemoryPressure时，不再调度新的BestEffort Pod；当节点为DiskPressure时，不再向节点调度Pod# kubernets原生GPU管理

### 两种机制

- Extend Resources：允许用户自定义资源名称。度量是整数级别；是一种Node级别的api，只需要通过PATCH API对Node对象的status部分更新即可
- Device Plugin Framework允许第三方设备提供商以外置方式对设备进行全生命周期管理，Device Plugin Framework建立Kubernetes和Devices Plugin之间的桥梁，负责设备信息上报和调度选择；每个硬件都需要Device Plugin进行管理，通过GRPC于kubelet的Device Plugin Manager进行连接，

需要注意的是 kubelet 在向 api-server 进行汇报的时候，只会汇报该 GPU 对应的数量。而 kubelet 自身的 Device Plugin Manager 会对这个 GPU 的 ID 列表进行保存，并用来具体的设备分配。而这个对于 Kubernetes 全局调度器来说，它不掌握这个 GPU 的 ID 列表，它只知道 GPU 的数量。

这就意味着在现有的 Device Plugin 工作机制下，Kubernetes 的全局调度器无法进行更复杂的调度。比如说想做两个 GPU 的亲和性调度，同一个节点两个 GPU 可能需要进行通过 NVLINK 通讯而不是 PCIe 通讯，才能达到更好的数据传输效果。在这种需求下，目前的 Device Plugin 调度机制中是无法实现的。

## 社区方案

---

# 社区的异构资源调度方案

- Nvidia GPU Device Plugin : <https://github.com/NVIDIA/k8s-device-plugin>
- GPU Share Device Plugin :
  - <https://github.com/AliyunContainerService/gpushare-scheduler-extender>
  - <https://github.com/AliyunContainerService/gpushare-device-plugin>
- RDMA Device Plugin : <https://github.com/Mellanox/k8s-rdma-sriov-dev-plugin>
- FPGA Device Plugin : [https://github.com/Xilinx/FPGA\\_as\\_a\\_Service/tree/master/k8s-fpga-device-plugin/trunk](https://github.com/Xilinx/FPGA_as_a_Service/tree/master/k8s-fpga-device-plugin/trunk)

## 1. GPU共享调度-阿里

按GPU的Memory去切分，多个任务可以调度到同一个卡上

<https://github.com/AliyunContainerService/gpushare-scheduler-extender>

```
apiVersion: apps/v1beta1
kind: StatefulSet

metadata:
 name: binpack-1
 labels:
 app: binpack-1

spec:
 replicas: 3
 serviceName: "binpack-1"
 podManagementPolicy: "Parallel"
 selector: # define how the deployment finds the pods it manages
 matchLabels:
 app: binpack-1

 template: # define the pods specifications
 metadata:
 labels:
 app: binpack-1

 spec:
 containers:
 - name: binpack-1
 image: cheyang/gpu-player:v2
 resources:
 limits:
 # GiB
 aliyun.com/gpu-mem: 3
```

## 2. Volcano device plugin

<https://github.com/volcano-sh/devices>