Kubernetes 基础

一、学习目标

1. 理解 Pod，掌握 Pod 的生命周期
2. 各种控制器类型的特点及使用定义方式
3. 掌握 svc 原理及其构建方式
4. 掌握多种存储类型的特点及其使用场景
5. 掌握调度器原理，根据要求把 Pod 定义到想要的节点运行
6. kubernetes 高可用集群搭建，kubeadm方式，二进制方式，**rancher方式**
7. 集群安全，认证-鉴权-访问控制原理及其流程
8. 掌握 Helm 原理及其使用

二、Kubernetes 基本概念

1、Kubernetes 简介

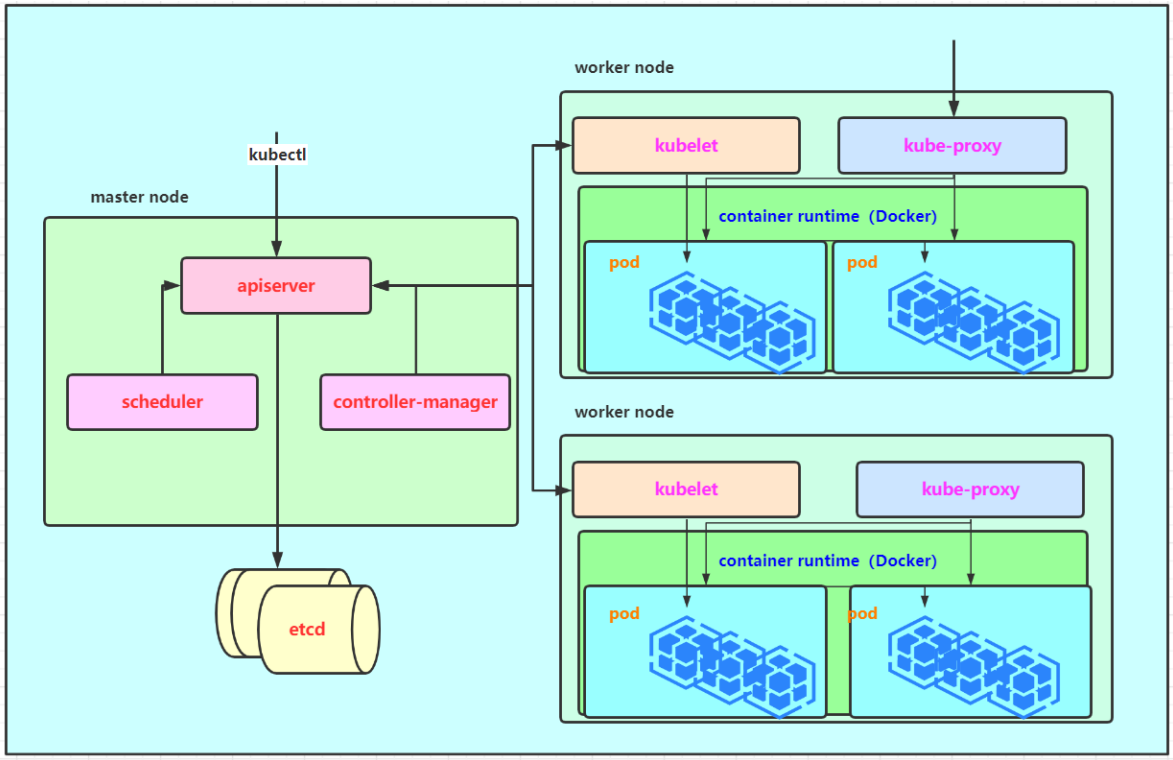
1. kubernetes 前世今生及特征

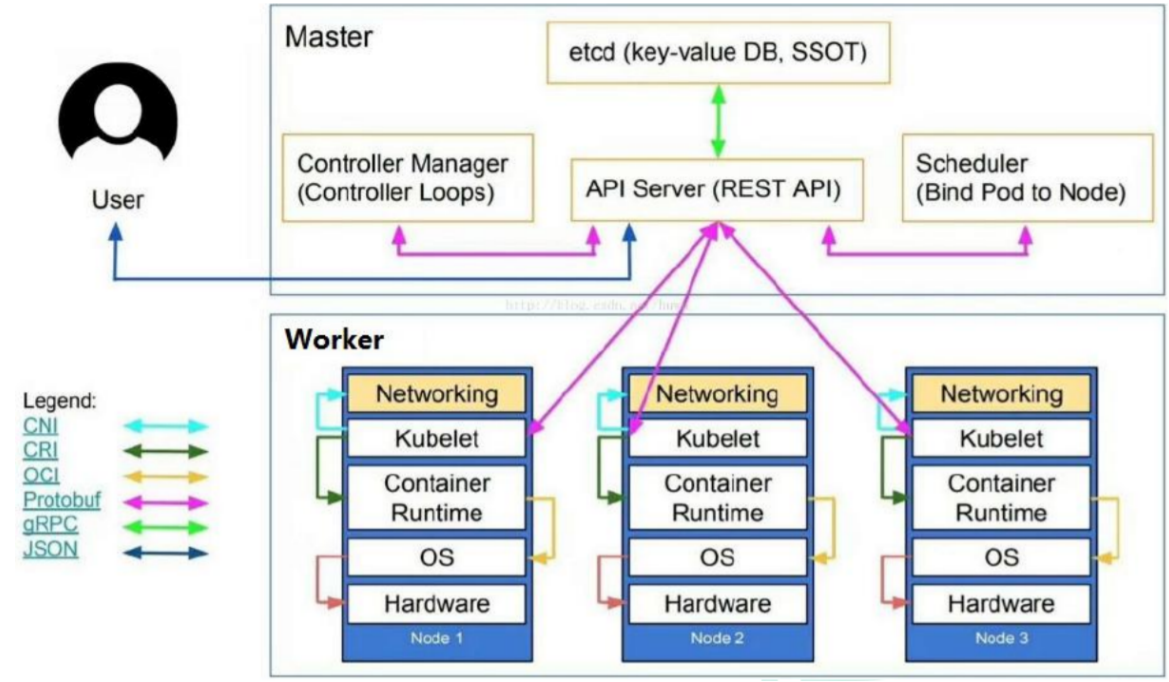
- 参考谷歌的 borg 设计而来，Go 语言开发
- 轻量级，消耗资源小
- 开源
- 弹性伸缩
- 负载均衡 IPVS
- 滚动更新
- 版本回退

2. kubernetes 概述

- k8s 是谷歌在2014年开源的容器化集群管理系统
- 使用 k8s 进行容器化应用部署
- 使用 k8s 利于应用扩展
- k8s 目标实施让部署容器化应用更加简洁和高效

2、K8S 集群架构组件





1、Master 组件（主控节点）

- **apiserver**
  集群统一入口，以 restful 方式，交给 Etcd 存储，所有服务的统一入口
- **scheduler**
  节点调度，选择合适的 node 节点应用部署
- **controller-manager**
  处理集群中常规后台任务，一个资源对应一个控制器，维持副本期望数目
- **etcd**
  兼具一致性和高可用性的键值数据库，用于保存 Kubernetes 集群相关的数据

2、Node 组件（工作节点）

- **kubelet**
  master 派到 node 节点代表，管理本机容器，保证容器都运行在 Pod 中，直接与容器引擎交互实现容器的生命周期管理

- **kube-proxy**
  提供网络代理，负载均衡等操作，负责写入规则至 iptables、ipvs 实现服务映射访问
- **Container Runtime**（容器运行环境）
  容器运行环境是负责运行容器的软件，支持的容器运行环境有： Docker、 containerd、cri-o、rktlet 以及任何实现 Kubernetes CRI (容器运行环境接口)

3、其他组件

- coreNDS
  可以为集群中的 svc 创建一个域名 ip 的对应关系解析
- dashboard
  给 k8s 集群提供一个 B/S 架构访问体系
- **ingress controller**
  官方只能实现四层代理，ingress 可以实现七层代理
- federation
  提供一个可以跨集群中心多 k8s 统一管理功能
- **Prometheus**
  提供 k8s 集群的监控能力
- **ELK**
  提供 k8s 集群日志统一分析平台

4. 运行机制

- 健康检查 + 重启策略实现了 Pod 故障自我修复能力
- 调度算法找到合适的节点将 Pod 分布式部署
- 控制器监控 Pod 的预期副本数，并根据 Node 失效状态自动在正常 Node 启动 Pod
- HPA 实现了 Pod 动态扩缩容

三、kubernetes 核心概念

0、资源清单

1. 资源清单格式

```
apiVersion: group/apiversion # 版本信息
kind:  # 资源类型
metadata:  # 资源元数据
  name:  # 资源名称
  namespace: # 名称空间，默认default
  labels:  # 资源标签 - 用于被筛选
  annotations:
spec:  # kubernetes 对象描述
status:  # 当前状态，本字段由 kubernetes 自身维护
```

```
# 获取 apiversion 版本信息
kubectl api-versions
# 获取资源的 apiversion 版本信息
kubectl explain pod Ingress
# 获取帮助文档
kubectl --help
kubectl get --help
```

2. 资源清单常用字段解释

参数名	字段类型	说明
apiVersion	String	这里是指 k8s api 的版本，目前基本是 v1，使用 kubectl api-versions 查看
kind	String	资源类型和角色，Pod，Deployment等
metadata	Object	元数据对象，固定值
metadata.name	String	元数据对象名称，用户自定义
metadata.namespace	String	元数据对象命名空间，默认default，可以自定义
spec	Object	详细定义对象，固定值 spec
spec.containers[]	List	spec对象的容器列表定义
spec.containers[].name	String	容器名称
spec.containers[].image	String	容器镜像名称
spec.containers[].imagePullPolicy	String	镜像拉取策略，Always、Never、IfNotPresent
spec.containers[].command[]	List	容器启动命令
spec.containers[].args[]	List	容器启动参数
spec.containers[].livenessProbe	Object	容器存活探针
spec.containers[].readinessProbe	Object	容器就绪探针
spec.containers[].startupProbe	Object	容器启动探针
spec.containers[].workingDir	String	指定容器的工作目录
spec.containers[].volumeMounts[]	List	指定容器内部的存储卷配置
spec.containers[].volumeMounts[].name	String	指定可以被容器挂载的存储卷的名称
spec.containers[].volumeMounts[].mountPath	String	指定可以被容器挂载的存储卷的路径
spec.containers[].volumeMounts[].readOnly	String	设置存储卷路径的读写模式，true或false，默认false读写模式
spec.containers[].ports[]	List	指定容器需要用到的端口列表
spec.containers[].ports[].name	String	指定端口名称
spec.containers[].ports[].containerPort	String	指定容器需要监听的端口号

参数名	字段类型	说明
spec.containers[].ports[].hostPort	String	指定容器所在宿主机监听的端口号，默认跟 containerPort相同，注意多个指定了 hostPort 的端口号不能重复
spec.containers[].ports[].protocol	String	指定端口协议，支持 TCP 和 UDP，默认 TCP
spec.containers[].env[]	List	指定容器运行前需设置的环境变量列表
spec.containers[].env[].name	String	指定环境变量名称
spec.containers[].env[].value	String	指定环境变量值
spec.containers[].resources	Object	指定资源限制和资源请求的值
spec.containers[].resources.limits	Object	指定容器运行时资源的运行上限
spec.containers[].resources.limits.cpu	String	指定CPU的最大限制，单位为 core 数
spec.containers[].resources.limits.memory	String	指定内存的最大限制，单位为 MIB、GiB
spec.containers[].resources.requests	Object	指定容器启动和调度时的资源限制
spec.containers[].resources.requests.cpu	String	容器启动和调度时CPU限制
spec.containers[].resources.requests.memory	String	容器启动和调度内存限制
spec.restartPolicy	String	Pod 重启策略，Always、OnFailure、Never
spec.nodeSelector	Object	定义Node的Label 过滤标签，key-value格式
spec.imagePullSecrets	Object	定义拉取镜像时使用secret名称，以 name:secretkey格式指定
spec.hostNetwork	Boolean	定义是否使用主机网络模式，默认false，设置true表示使用宿主机网络，不使用docker网桥，同时无法在同一台宿主机上启动第二个副本

1、Pod

1. Pod 基本概念

- 最小部署的单元
- 包含多个容器（一组容器的集合，一个或多个容器）

- 一个 pod 中的容器共享网络命名空间
- pod 是短暂的

2. Pod 存在意义

- 创建容器使用 docker，一个 docker 对应一个容器，一个容器有进程，一个容器运行一个应用程序
- Pod 是多进程设计，运行多个应用程序
  一个 Pod 有多个容器，一个容器里面运行一个应用程序
- Pod 存在为了亲密性应用
- 两个应用之间进行交互
- 网络之间调用
- 两个应用需要频繁调用

3. Pod 实现机制

- 共享网络
  共享网络：通过 Pause 容器，把其他业务容器加入到 Pause 容器里面，让所有业务容器在同一个名称空间中，可以实现网络共享
- 共享存储
  共享存储：引入数据卷概念 Volumn，使用数据卷仅限持久化存储

4. Pod 镜像拉取策略

拉取策略	说明
IfNotPresent	默认值，镜像在宿主机不存在才拉取
Always	每次创建 Pod 都会重新拉取一次镜像
Never	Pod 永远不会主动拉取镜像，需要开发人员手动拉取镜像

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: nginx
    image: nginx:1.14
    imagePullPolicy: Awalys
```

5. Pod 资源限制（resources）

资源限制	说明
requests	调度资源限制
limits	最大资源限制

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: fronted
spec:
  containers:
```

```
  - name: db
    image: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "password"
    resources:
      requests:
        memery: "64Mi"
        cpu: "250m"
      limits:
        memery: "128Mi"
        cpu: "500m"
```

6. Pod 重启策略（restartPolicy）

重启策略	说明
Always	当容器终止退出后，总是重启容器，默认策略
OnFailure	当容器异常退出（退出状态码非0）时，才重启容器
Never	当容器终止退出，从不重启容器

```
apiVersion: v1
kind: Pod
metadata:
  name: dns-test
spec:
  containers:
  - name: busybox
    image: busybox:1.28.4
    args:
    - /bin/sh
    - -c
    - sleep 36000
 restartPolicy: Never
```

7. Pod 容器配置健康检查

探针	说明
livenessProbe	如果检查失败，将杀死容器，根据 Pod 的 重启策略 restartPolicy 来操作，如果容器不提供存活探针， 则默认状态为 `Success`
readinessProbe	如果检查失败，kubernetes 会把 Pod 从 service endpoints 中剔除，如果容器不提供就绪态探针，则默认状态为 `Success`
startupProbe	指示容器中的应用是否已经启动。如果提供了启动探针，则所有其他探针都会被禁用，直到此探针成功为止，如果启动探测失败，`kubelet` 将杀死容器，而容器依其 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 `Success`

- Probe 探针支持的三种检查方法：

方法	说明
httpGet	发送 HTTP 请求，返回 200-400 范围状态码为成功
exec	执行 shell 命令返回状态码为0为成功
tcpSocket	发起 TCP Socket 建立成功

```
apiVersion: v1
kind: Pod
metadata:
  # Pod 名称
  name: readiness-check
  # 命名空间
  namespace: default
spec:
  containers:
  # 容器名称
  - name: readiness-container
    # 容器镜像
    image: nginx
    # 镜像拉取策略
    imagePullPolicy: IfNotPresent
    #command: ["/bin.sh","-c","touch /tmp/live; sleep 60; rm -rf /tmp/live;
sleep 3600"]
    # 就绪探测
    readinessProbe:
      # 1. httpGet 方式
      httpGet:
        port: 80
        path: /index.html
      initialDelaySeconds: 1
      periodSeconds: 3
      # 2. exec 方式
      #exec:
        #command: ["test","-e","/tmp/live"]
      # 3. tcpSocket 方式
      #tcpSocket:
        #port: 80
```

- `lifecycle` 容器启动、退出动作
  - postStart
  - preStop

```
apiVersion: v1
kind: Pod
metadata:
  name: lifecycle
spec:
  containers:
  - name: lifecycle-container
    image: nginx
     # lifecycle
    lifecycle:
     # postStart
      postStart:
```

```
        exec:
          command: ["/bin/sh","-c","echo hello start > /usr/local/message"]
      # preStop
       preStop:
        exec:
          command: ["/bin/sh","-c","echo hello stop > /usr/local/message"]
```

8. Init 容器

- 是一种特殊容器
- 在 Pod 内的应用容器启动之前运行
- Init 容器与普通的容器非常像
- 它们总是运行到完成
- 每个都必须在下一个启动之前成功完成
- Init 容器失败，kubelet 会不断地重启直到成功，除非 Pod 的重启策略restartPolicy 为 Never
- Init 容器不支持 `lifecycle`、`livenessProbe`、`readinessProbe` 和 `startupProbe`

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
   containers:
   # 应用容器
   - name: myapp-container
     image: busybox:1.28
     command: ['sh', '-c', 'echo The app is running! && sleep 3600']
   # Init 容器
   initContainers:
   - name: init-myservice
     image: busybox:1.28
     command: ['sh', '-c', "until nslookup myservice.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do
echo waiting for myservice; sleep 2; done"]
   - name: init-mydb
     image: busybox:1.28
     command: ['sh', '-c', "until nslookup mydb.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do
echo waiting for mydb; sleep 2; done"]
---
apiVersion: v1
kind: Service
metadata:
  name: myservice
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
---
apiVersion: v1
kind: Service
metadata:
  name: mydb
spec:
```

```
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9377
```

9. Pod 分类

类型	说明
自主式 Pod	Pod 退出了，此类型的 Pod 不会被创建
控制器管理的 Pod	在控制器的生命周期里，始终要维持 Pod 的副本数目

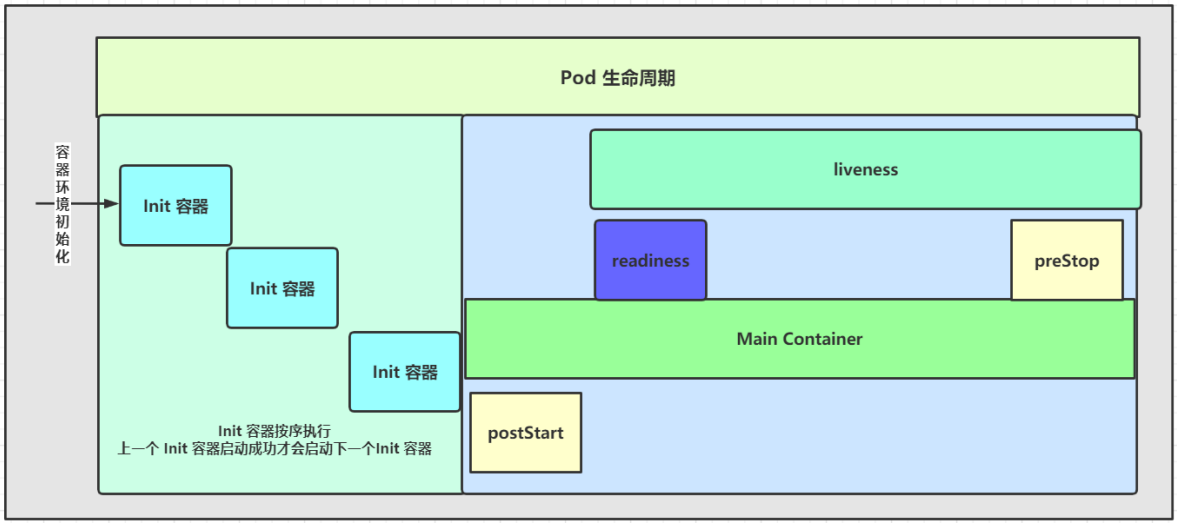
10. Pod 的状态

状态值	说明
Pending	Pod 已被 Kubernetes 系统接受，但有一个或者多个容器尚未创建亦未运行。此阶段包括等待 Pod 被调度的时间和通过网络下载镜像的时间
Running	Pod 已经绑定到了某个节点，Pod 中所有的容器都已被创建。至少有一个容器仍在运行，或者正处于启动或重启状态
Succeeded	Pod 中的所有容器都已成功终止，并且不会再重启
Failed	Pod 中的所有容器都已终止，并且至少有一个容器是因为失败终止。也就是说，容器以非 0 状态退出或者被系统终止
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

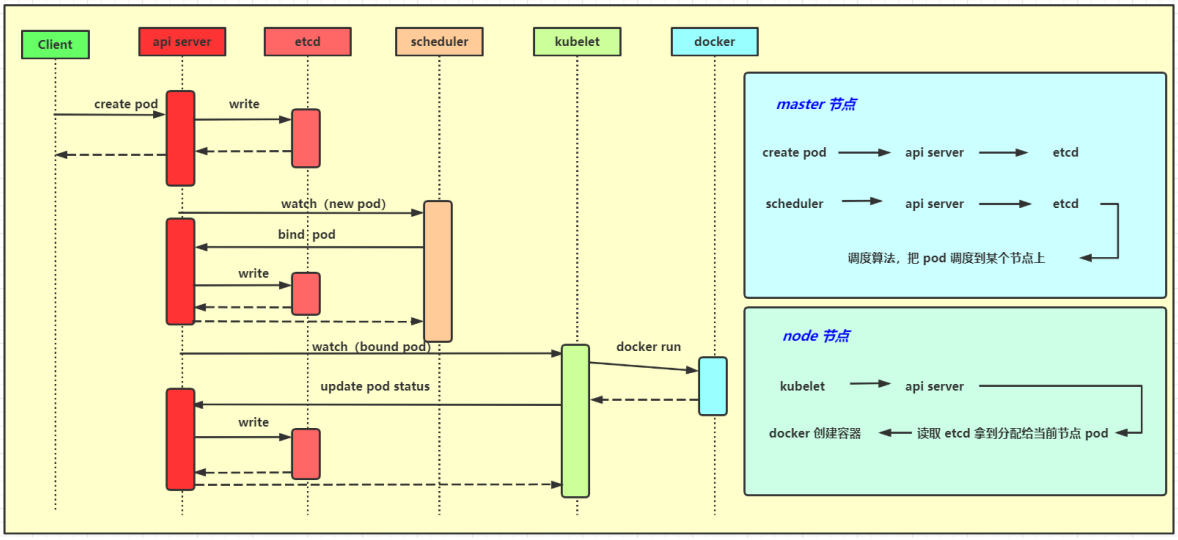
容器状态

- Waiting 等待
- Running 运行中
- Terminated 已终止

11. 容器生命周期



12. 创建 Pod 的流程



13. 影响 Pod 调度配置（Scheduler）

1. Pod 资源限制影响

> 根据 requests 找到足够 node 节点进行调度

```
....
resources:
# requests 资源限制
   requests:
     memery: "64Mi"
     cpu: "250m"
```

2. 节点选择器标签影响

- 首先对节点创建标签

```
# 节点创建标签 kubectl label nodes <node-name> <label-key>=<label-value>
kubectl label nodes node01 env_role=dev
# 查看节点标签
kubectl get nodes --show-labels
```

- 添加 nodeSelector 字段到 Pod 配置中

```
....
 spec:
  # 节点选择器标签
  nodeSelector:
   env_role: dev
  containers:
  - name: nginx
   image: nginx:1.15
```

3. 节点亲和性影响

> 节点亲和性 nodeAffinity 和前面的 nodeSelector 基本一样，根据节点上标签约束来决定 Pod 调度到哪些节点上
>
> - 硬亲和性：约束条件必须满足 requiredDuringSchedulingIgnoredDuringExecution
> - 软亲和性：尝试满足，不保证 preferredDuringSchedulingIgnoredDuringExecution

支持常用操作符：

In、NotIn、Exists、Gt、Lt、DoesNotExists

```
# 节点创建标签
kubectl label nodes node01 kubernetes.io/e2e-az-name=e2e-az1
```

```
apiVersion: v1
kind: Pod
metadata:
 name: with-node-affinity
spec:
 affinity:
    nodeAffinity:
      # 硬亲和性
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
       # 软亲和性
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: another-node-label-key
            operator: In
            values:
            - another-node-label-value
 containers:
  - name: with-node-affinity
    image: k8s.gcr.io/pause:2.0
```

4. 污点和污点容忍影响

基本介绍：

- 污点：节点亲和性（详见这里） 是 Pod 的一种属性，它使 Pod 被吸引到一类特定的节点。这可能出于一种偏好，也可能是硬性要求。 Taint（污点）则相反，它使节点能够排斥一类特定的 Pod
- 容忍度（Tolerations）是应用于 Pod 上的，允许（但并不要求）Pod 调度到带有与之匹配的污点的节点上

场景：

- 专用节点
- 配置特点硬件节点
- 基于 Taint 驱逐

操作演示：

- 查看节点污点情况

```
kubectl describe node k8smaster | grep Taint
```

> 污点值有三个：
>
> 1. NoSchedule：一定不被调度
> 2. PreferNoSchedule：尽量不被调度
> 3. NoExecute：不会调度，并且还会驱逐Node已有的 Pod

- 为节点添加污点

```
# kubectl taint node [node] key=value:污点值
kubectl taint node k8smaster env_role=dev:NoSchedule
```

- 删除污点

```
kubectl taint node k8smaster env_role=dev:NoSchedule-
```

- 污点容忍

```
apiVersion: v1
kind: Pod
metadata:
name: nginx
labels:
   env: test
spec:
containers:
  - name: nginx
   image: nginx
   imagePullPolicy: IfNotPresent
   # 污点容忍，可以容忍节点存在example-key=情况，除非还有其他污点
tolerations:
  - key: "example-key"
   operator: "Exists"
   effect: "NoSchedule"
```

方便理解案例如下：

- 假设节点添加了三个污点

```
kubectl taint nodes node1 key1=value1:NoSchedule
kubectl taint nodes node1 key1=value1:NoExecute
kubectl taint nodes node1 key2=value2:NoSchedule
```

- pod 有2个污点容忍

```
....
tolerations:
  - key: "key1"
   operator: "Equal"
   value: "value1"
   effect: "NoSchedule"
  - key: "key1"
   operator: "Equal"
   value: "value1"
   effect: "NoExecute"
```

> 结论：这个 pod 可以容忍2个污点，但是容忍度没有匹配第三个污点，因此该 pod 不会分配到当前节点

2、Controller

1. Controller 简介

1. 是什么?

集群上管理和运行容器的对象

2. Pod 和 Controller 关系

- Pod 是通过 Controller 实现应用的运维，比如伸缩，滚动升级等
- Pod 和 Controller 之间通过 label 标签建立关系 => selector

3. Deployment 控制器应用场景

- 部署无状态应用
- 管理 Pod 和 ReplicaSet
- 部署、滚动升级等功能 => web服务、微服务

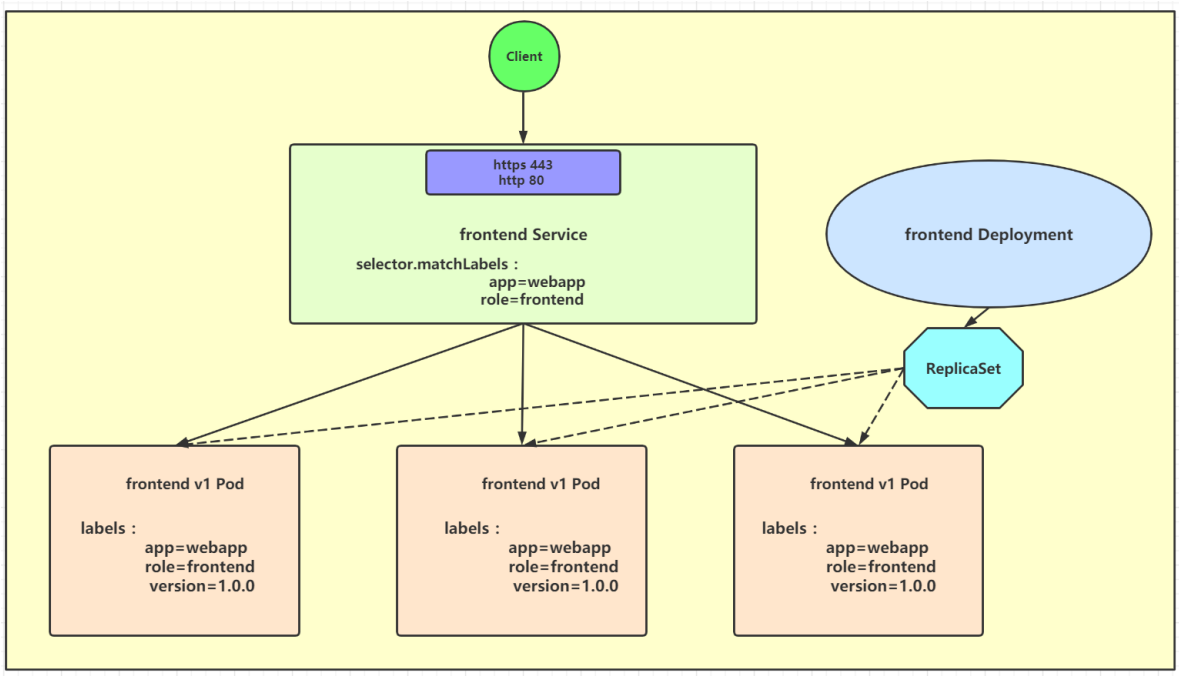
4. 基本操作

```
# 导出 yaml 文件
kubectl create deployment web --image=nginx --try-run -o yaml > web.xml
# 使用 yaml 部署应用
kubectl apply -f web.yaml
# 对外发布（暴露对外端口号）
kubectl expose deployment web --port=80 --type=NodePort --target-port=80 --
name=web1 -o yaml > web1.xml
kubectl apply -f web1.yaml
# 查看pod 和 svc 状态
kubectl get pods,svc
# 应用升级
kubectl set image deployment web nginx=nginx:1.15
# 查看升级状态
kubectl rollout status deployment web
# 查看升级版本
kubectl rollout history deployment web
# 回滚到上一个版本
kubectl rollout undo deployment web
# 回滚到指定版本
kubectl rollout undo deployment web --to-revision=2
# 弹性伸缩
kubectl scale deployment web --replicas=10
```

2. 控制器类型

类型	说明
ReplicationController	RC在新版本已被ReplicaSet取代，用来确保用户定义的副本数，即容器异常退出会自动创建新的Pod来替代
ReplicaSet	和 RC 没有本质区别， RS 支持集合式的 selector
Deployment	为 Pod 和 RS 提供了一个声明式定义方法，定义Deployment 来创建 Pod 和 RS，滚动升级和回滚应用，扩缩容，暂停/继续 Deployment
DaemonSet	确保全部 Node 上运行一个 Pod 副本，有 Node 加入集群时为该 Node新增一个 Pod，移除 Node 时回收 Pod，日志收集，监控等
StatefulSet	为 Pod 提供唯一的标识，保证部署和 scale 的顺序，解决有状态服务的问题
Job/CronJob	Job 负责批处理任务，只执行一次，CronJob 管理基于时间的 Job，在给定时间点只运行一次，周期性的运行
Horizontal Pod Autoscaling	HPA 使得 Pod 水平自动缩放，削峰填谷，提高系统资源利用率

3. RS 与 Deployment 的关系



4. 控制器 yaml 示例

① ReplicationController

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  # 副本数
  replicas: 3
  # 标签选择符，不配置默认和 .template.metadata.labels保持一致
  # 若配置则和 .template.metadata.labels 进行匹配，如果不匹配则API拒绝
  # 与 RS 的区别是 selector 这里只支持单值，RS 支持集合
```

```
  selector:
    app: nginx
  # Pod 模板
  template:
    metadata:
      name: nginx
      # Pod 标签
      labels:
        app: nginx
    spec:
      # 容器
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

基本操作

```
# 运行示例任务，可以本地创建 yaml 文件
kubectl apply -f https://k8s.io/examples/controllers/replication.yaml
# 检查 ReplicationController 的状态
kubectl describe replicationcontrollers/nginx
# 删除 RC 及其 Pod
kubectl delete -f https://k8s.io/examples/controllers/replication.yaml
# 只删除 RC
kubectl delete -f https://k8s.io/examples/controllers/replication.yaml --
cascade=false
# 修改 Pod 标签可以将 Pod 从 RC 中隔离
```

② ReplicaSet

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  # RS 标签
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  # selector.matchLabels - 与 ReplicationController 的 .spec.selector 的作用相同
  # matchExpressions 允许构建更加复杂的选择器
  # 可以通过指定 key、value 列表以及将 key 和 value 列表关联起来的 operator
  # RS 标签选择符与 .template.metadata.labels 进行匹配
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      # Pod 标签
      labels:
        tier: frontend
    spec:
       containers:
```

```
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
```

基本操作

```
# 创建 rs
kubectl apply -f https://kubernetes.io/examples/controllers/frontend.yaml
# 查看 rs
kubectl get rs
# 查看 rs 状态
kubectl describe rs/frontend
# 查看启动的 pod
kubectl get pods
# 以 yaml 格式查看 Pod 的
kubectl get pods frontend-b2zdv -o yaml
```

③ Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  # deployment 标签
  labels:
    app: nginx
spec:
  # Pod 副本数,默认 1
  replicas: 3
  # deployment 标签选择符，选择与之匹配的 Pod
  selector:
    matchLabels:
      app: nginx
  # Pod 模板
  template:
    metadata:
     # Pod 标签
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

基本操作

```
# 创建 Deployment
kubectl apply -f https://k8s.io/examples/controllers/nginx-deployment.yaml --
record
# 查看 Deployment
kubectl get deployments
# 查看 Deployment 上线状态
kubectl rollout status deployment/nginx-deployment
# 查看 Deployment 创建的 ReplicaSet
kubectl get rs
```

```
# 查看每个 Pod 自动生成的标签
kubectl get pods --show-labels
# 更新 Deployment
kubectl --record deployment.apps/nginx-deployment set image
  deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1
# 或者
kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1 --record
# 或者 edit Deployment 的 .spec.template.spec.containers[0].image 从 nginx:1.14.2
更改至 nginx:1.16.1
kubectl edit deployment.v1.apps/nginx-deployment
# 获取 Deployment 的更多信息
kubectl describe deployments
# 检查 Deployment 修订历史
kubectl rollout history deployment.v1.apps/nginx-deployment
# 查看修订历史的详细信息
kubectl rollout history deployment.v1.apps/nginx-deployment --revision=2
# 撤消当前上线并回滚到前一个的修订版本
kubectl rollout undo deployment.v1.apps/nginx-deployment
# 回滚到特定修订版本
kubectl rollout undo deployment.v1.apps/nginx-deployment --to-revision=2
# 缩放 Deployment
kubectl scale deployment.v1.apps/nginx-deployment --replicas=10
# Pod 的水平自动缩放 HPA
kubectl autoscale deployment.v1.apps/nginx-deployment --min=10 --max=15 --cpu-
percent=80
# 暂停、恢复 Deployment
kubectl rollout pause deployment.v1.apps/nginx-deployment
kubectl rollout resume deployment.v1.apps/nginx-deployment
# 观察上线状态
kubectl get rs -w
# 以 yaml 格式查看 Deployment
kubectl get deployment nginx-deployment -o yaml
# kubectl rollout 命令的退出状态为 1（表明发生了错误）
kubectl rollout
echo $?
```

④ DaemonSet

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  # 标签选择符
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  # pod 模板
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      # 容忍度
```

```
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        # 资源限制
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        # 挂载卷
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      # 挂载卷定义
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

基本操作

```
# 创建 daemonSet
kubectl apply -f https://k8s.io/examples/controllers/daemonset.yaml
```

⑤ **StatefulSet**

> 用来管理有状态应用的工作负载 API 对象
>
> 为 Pod 提供持久存储和持久标识符
>
> StatefulSet 为它们的每个 Pod 维护了一个有粘性的 ID
>
> 无论怎么调度，每个 Pod 都有一个永久不变的 ID
>
> StatefulSet 当前需要 无头服务 来负责 Pod 的网络标识
>
> deployment 和 statefulSet 区别：statefulSet 的每一个 Pod 有唯一标识
>
> 唯一标识格式：主机名称.service名称.名称空间.svc.cluster.local

参考 mysql部署

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
```

```yaml
spec:
  ports:
  - port: 80
    name: web
  # 无头服务
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

基本操作

```bash
# 创建
kubectl apply -f web.yaml
# 查看 service
kubectl get service nginx
# 查看 statefulSet
kubectl get statefulset web
# 查看pod
kubectl get pods -w -l app=nginx
# 删除 StatefulSet 中所有的 Pod
kubectl delete pod -l app=nginx
# 查看持久卷
kubectl get pvc -l app=nginx
# 扩容
```

```
kubectl scale sts web --replicas=5
# 缩容
kubectl patch sts web -p '{"spec":{"replicas":3}}'
```

StatefulSet 的启停顺序

> - 有序部署：部署 StatefulSet时，如果有多个 Pod 副本，它们会被顺序地创建（从0到N-1），并且，在下一个Pod 运行之前所有之前的 Pod 必须都是 Running 和 Ready 状态
> - 有序删除：当 Pod 被删除时，它们被终止的顺序是从 N-1 到 0
> - 有序扩展：当对 Pod 执行扩展操作时，与部署一样，它前面的 Pod 必须都处于 Running 和 Ready 状态
>
> StatefulSet 的使用场景
>
> - 稳定的持久化持仓，即Pod重新调度后还是能访问到相同的持久化数据，基于 PVC 实现
> - 稳定的网络标识，即Pod重新调度后其 PodName 和 HostName 不变
> - 有序部署，有序扩展，基于 init containers 来实现
> - 有序收缩

⑥ Job

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

基本操作

```
# 运行 job
kubectl apply -f https://kubernetes.io/examples/controllers/job.yaml
# 检查 Job 的状态
kubectl describe jobs/pi
# 查看其中一个 Pod 的标准输出
kubectl logs $pods
```

⑦ CronJob

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
```

```
          - name: hello
            image: busybox
            imagePullPolicy: IfNotPresent
            command:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

基本操作

```
# 运行 CronJob
kubectl create -f https://k8s.io/examples/application/job/cronjob.yaml
# 获取 CronJob 状态
kubectl get cronjob hello
# 监视这个任务
kubectl get jobs --watch
# 查看 Pod 日志
kubectl logs $pods
# 删除 CronJob
kubectl delete cronjob hello
```

3、Service

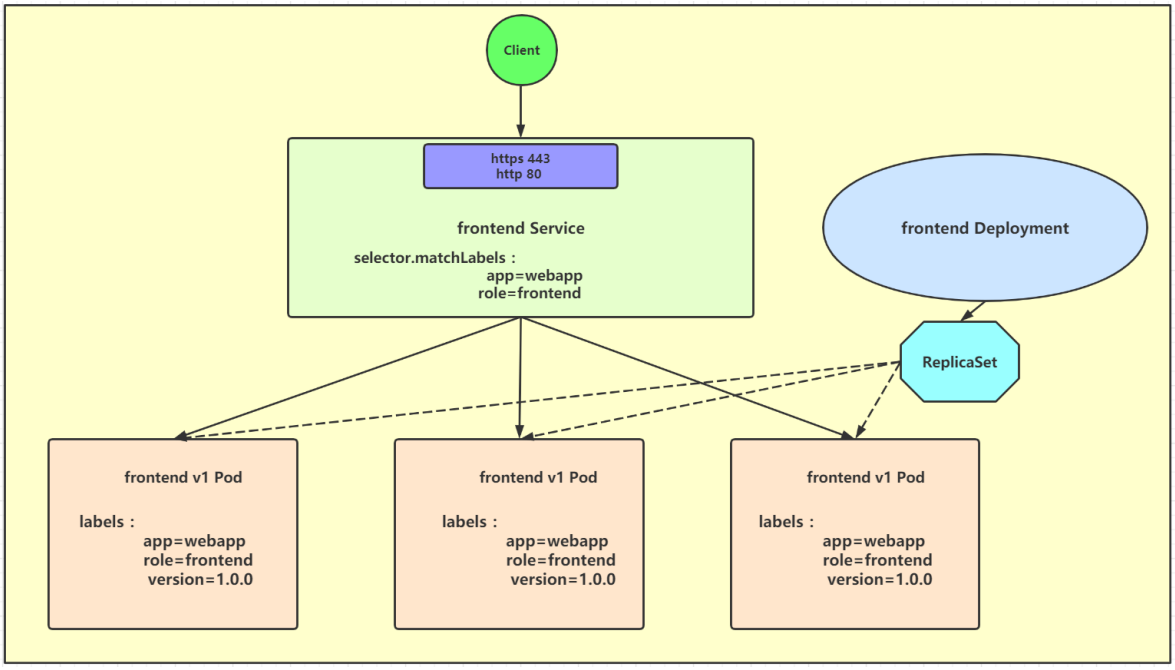
1、Service 简介

1. Service 存在的意义

- 防止 Pod 失联（服务发现）
- 定义一组 Pod 访问策略（负载均衡），一般根据 label 和 selector 标签与 Pod 建立关联
- 只提供 4 层负载均衡能力，Ingress 提供 7 层

2. Pod 和 Service 关系

Service 定义了一种抽象，是一个 Pod 的逻辑分组，一种访问 Pod 的策略，这一组 Pod 能够被 Service 访问到，通过 label selector 与 Pod 建立关联，Service 对外暴露服务，提供 Pod 服务代理



3. Service 常用类型

- ClusterIp： 默认类型，自动分配一个仅 Cluster 内部可以访问的虚拟 IP
- NodePort： 在 ClusterIP 基础上为 Service 在每台机器上绑定一个端口，这样就可以通过 NodePort 来访问该服务
- LoadBalancer： 在 NodePort 基础上，借助 cloud provider 创建一个外部负载均衡器，并将请求转发到 NodePort
- ExternalName： 把集群外部的服务引入到集群内部来，在集群内部直接使用，kubernetes1.7 及以上版本支持

4. 代理模式分类

- userspace : 造成 api-server 访问压力过大
- iptables : 支持常见的负载均衡算法
- ipvs： 高版本默认，比 iptables 支持更多的负载均衡算法

2、Service yaml 示例

```yaml
# Service 定义
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  # Service 类型，默认ClusterIp
  type: NodePort
  # 端口
  ports:
  # 对外端口 8080
  - port: 8080
    # 转发端口 80
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    protocol: TCP
    name: https
    # 标签选择器，匹配 Pod 标签
  selector:
    run: my-nginx
---
# 定义 控制器
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 1
  # Pod 模板
  template:
    metadata:
    # Pod 标签
      labels:
        run: my-nginx
```

```
    spec:
    # 挂载卷定义
      volumes:
      - name: secret-volume
        secret:
          secretName: nginxsecret
      - name: configmap-volume
        configMap:
          name: nginxconfigmap
      containers:
      - name: nginxhttps
        image: bprashanth/nginxhttps:1.0
        ports:
        - containerPort: 443
        - containerPort: 80
        # 挂载卷使用
        volumeMounts:
        - mountPath: /etc/nginx/ssl
          name: secret-volume
        - mountPath: /etc/nginx/conf.d
          name: configmap-volume
```

```
# 创建 Service，Deployment，RS，Pod
kubectl create -f nginx-secure-app.yaml
# 查看 svc
kubectl get svc my-nginx
# 查看 svc 详情
kubectl describe svc my-nginx
```

4、Ingress

1. 是什么?

Ingress 公开了从集群外部到集群内服务的 HTTP 和 HTTPS 路由。 流量路由由 Ingress 资源上定义的规则控制，Ingress 是对集群中服务的外部访问进行管理的 API 对象，典型的访问方式是 HTTP，Ingress 可以提供负载均衡、SSL 和基于名称的虚拟托管

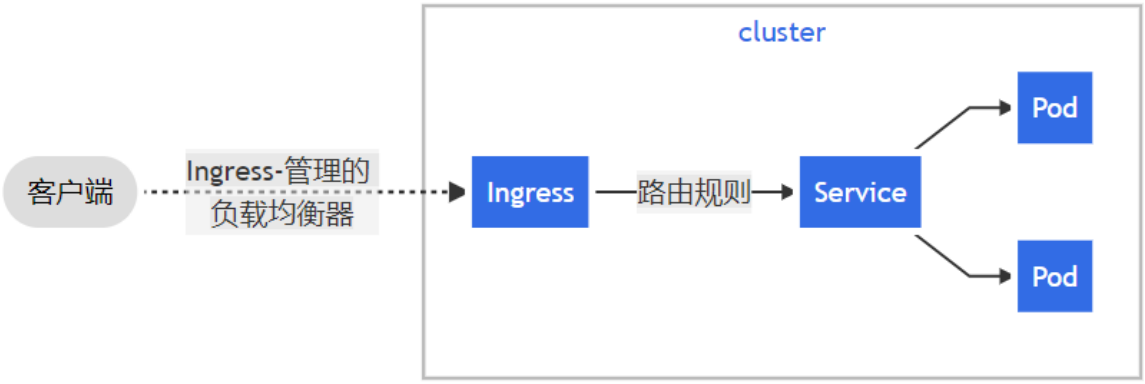
2. Service 实现及问题

1. 把端口号对外暴露，通过 ip + 端口号进行访问：使用 Service 的 NodePort 类型实现
2. NodePort 缺陷
   在每个节点上都会开启端口，在访问时候通过任何节点，通过节点 ip + 暴露端口号实现访问
   意味着每个端口号只能使用一次，一个端口号对应一个应用
   实际访问都是使用域名，根据不同域名跳转到不同端口服务中

3. Ingress 和 Pod 的关系

Pod 和 Ingress 通过 Service 关联
Ingress 作为统一入口，由 Service 关联一组 Pod

4. Ingress 工作流程



5. Ingress 的使用

1. 部署 Ingress Controller

```
# 创建 nginx 应用，对外暴露端口使用 NodePort
kubectl create deployment web --image=nginx
# 对外暴露端口
kubectl expose deployment web --port=80 --target-port=80 --type=NodePort
# 部署官方维护的nginx ingress controller
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-
nginx/controller-v0.46.0/deploy/static/provider/cloud/deploy.yaml
# 查看 ingress controller 状态
kubectl get pods -n ingress-nginx
```

2. 创建 Ingress 规则

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-myapp
  namespace: default
  annotations:
    kubernetes.io/ingess.class: "nginx"
spec:
  rules:
  - host: shadow.com
    http:
      paths:
      - path: /
        backend:
          serviceName: ingress-nginx-controller
          servicePort: 80   #注此处必须要和后端pod的service的端口一致，否则会报503错误
```

运行

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

3. hosts 文件配置配置

```
10.25.175.109  shadow.com
```

4. 访问服务

```
shadow.com
```

5、ConfigMap

1. ConfigMap 简介

ConfigMap 功能在Kubernetes1.2 版本中引入，许多应用程序会从配置文件、命令行参数或环境变量中读取配置信息。ConfigMap API 给我们提供了向容器中注入配置信息的机制，ConfigMap 可以被用来保存单个属性，也可以用来保存整个配置文件或者JSON 二进制大对象

2. ConfigMap 存在的意义

Pods 可以将其用作环境变量、命令行参数或者存储卷中的配置文件，存储不加密数据到 etcd

3. ConfigMap 的使用

- 创建配置文件

```
cd /usr/local/configmap
vim game.properties
```

输入以下内容

```
email=xxx@qq.com
lives=3
secret.passwd=xoxo
```

```
vim ui.properties
```

输入以下内容

```
color.good=green
color.bad=red
show.flag=false
```

- 使用文件创建 ConfigMap [--from-file]

```
kubectl create configmap game-config --from-file=/usr/local/configmap
```

- --from-file 指定目录下的所有文件都会被用在ConfigMap里面创建一个键值对，键的名字就是文件名，值就是文件的内容

```
# 指定一个配置文件创建 ConfigMap
kubectl create configmap game-config-2 --from-
file=/usr/local/configmap/ui.properties
# 查看
kubectl get configmaps game-config-2 -o yaml
```

- --from-file 可以使用多次，指定多个文件
- 使用字面值创建 ConfigMap [--from-literal]

```
kubectl create configmap my-config --from-literal=my.name=shadow --from-
literal=my.addr=张家界
# 查看
kubectl get configmaps my-config -o yaml
```

- Pod 中使用 ConfigMap
- 环境变量方式

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
  namespace: default
data:
  log_level: INFO
---
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  containers:
  - name: test-container
    image: nginx:1.15
    command: ["/bin/sh","-c","env"]
    env:
    - name: SPECIAL_LEVEL_KEY
      valueFrom:
        configMapKeyRef:
          name: special-config
          key: special.how
    - name: SPECIAL_TYPE_KEY
      valueFrom:
        configMapKeyRef:
          name: special-config
          key: special.type
  restartPolicy: Never
```

- 命令行参数方式

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
```

```
  special.type: charm
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
  namespace: default
data:
  log_level: INFO
---
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  containers:
  - name: test-container
    image: nginx:1.15
    command: ["/bin/sh","-c","echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)"]
    env:
    - name: SPECIAL_LEVEL_KEY
      valueFrom:
        configMapKeyRef:
          name: special-config
          key: special.how
    - name: SPECIAL_TYPE_KEY
      valueFrom:
        configMapKeyRef:
          name: special-config
          key: special.type
  restartPolicy: Never
```

- 数据卷方式

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
---
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  containers:
  - name: test-container
    image: nginx:1.15
    command: ["/bin/sh","-c","cat /etc/config/special.how"]
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
  volumes:
  - name: config-volume
```

```
    configMap:
        name: special-config
  restartPolicy: Never
```

- 执行命令

```
kubectl apply -f xxx.yaml
# 查看
kubectl get cm
kubectl describe cm xxx
kubectl get pods
kubectl logs xxx
```

6、Secret

1. Secret 存在的意义

ConfigMap 解决了配置明文存储的问题，Secret 解决了密码，token，秘钥等敏感信息数据的配置存储问题。 Secret可以以Volume或者环境变量的方式使用

2. Secret 类型

- Service Account
  用来访问 Kubernetes API ，由 Kubernetes 自动创建，并且会自动挂载到 Pod 的/run/secrets/kubernetes.io/serviceaccont 目录中
- **Opaque**
  base64 编码格式的Secret，用来存储密码，秘钥等
- kubernetes.io/dockerconfigjson
  用来存储私钥 docker registry 的认证信息

3. Secret 的使用

- Service Account

```
# 创建 deployment
kubectl run nginx --image=nginx
# 查看 pod
kubectl get pods
# 查看 pod 内部文件
kubectl exec nginx-xxx ls
```

- Opaque
  Opaque 类型的数据是一个 map 类型，要求 value 是 base64 编码格式

```
# 得到 base64 编码内容
echo -n "admin" | base64
# YWRtaW4=
echo -n "Ptmjygb@1002" | base64
# UHRtanlnYkAxMDAy
```

- 挂载卷的方式使用 Secret

```
apiVersion: v1
kind: Secret
metadata:
```

```
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: UHRtanlnYkAxMDAy
---
# Secret 挂载到 Volume 中
apiVersion: v1
kind: Pod
metadata:
  name: secret-test
  labels:
    name: secret-test
spec:
  volumes:
  - name: secrets
 # 放在卷中的 Secret
    secret:
      secretName: mysecret
   # 将 Secret 键名映射到特定路径
      items:
      - key: username
        path: /usr/local/my-username
  containers:
  - name: db
    image: nginx:1.15
    volumeMounts:
    - name: secrets
      mountPath: "/etc/secret"
      readOnly: true
```

- 环境变量的方式使用 Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: UHRtanlnYkAxMDAy
# immutable 为true创建不可更改的 Secret
  immutable: true
---
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: redis
    image: redis
    env:
    - name: SECRET_USERNAME
   # 环境变量方式使用 Secret
      valueFrom:
        secretKeyRef:
```

```
          name: mysecret
          key: username
    - name: SECRET_PASSWORD
      valueFrom:
        secretKeyRef:
          name: mysecret
          key: password
  restartPolicy: Never
```

- kubernetes.io/dockerconfigjson
  使用 Kubernetes 创建 docker registry 认证的 secret，访问docker私服 harbor 会使用到

```
kubectl create secret docker-registry myregistrykey --docker-
server=DOCKER_REGISTRY_SERVER --docker-name=DOCKER_USER --docker-
password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL secret "myregistrykey"
```

Pod 中使用 imagePullSecrets 指定

```
apiVersion: v1
kind: Pod
metadata:
  name: foo
spec:
  containers:
  - name: foo
    image: nginx
# 引用创建的 myregistrykey
  imagePullSecrets:
  - name: myregistrykey
```

7、Volume

1. Volume 概述

> Volume 是 Pod 中能够被多个容器访问的共享目录，Volume 与 Pod 生命周期相同，与容器生命周期无关，容器终止或重启时 Volume中的数据不会丢失
>
> k8s 支持多种类型的 Volume，常见的如下
>
> - emptyDir
>
> Pod 调度到宿主机上时创建，Pod 中的容器都能读写 emptyDir，一旦这个 Pod 离开这个宿主机，emptyDir 的数据会被永久删除 - 一般作为临时空间
>
> - hostPath
>
> 此类型使得对应的容器能够访问当前宿主机上的指定目录，Pod 离开这个宿主机时，hostPath 中的数据虽然不会被永久删除，但数据也不会随 Pod 迁移到其他宿主机上
>
> - nfs
>
> nfs 类型的 Volume 允许一块现有的网络磁盘在同一个 Pod 内的容器间共享

2. empthDir 的用法

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: test-empty-dir
spec:
  containers:
  - image: nginx:1.14
    name: nginx
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

3. hostPath 的用法(注意挂载卷的权限)

```
apiVersion: v1
kind: Pod
metadata:
  name: test-host-path
spec:
  containers:
  - image: nginx:1.15
    name: nginx
    volumeMounts:
    - mountPath: /test-pod
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
   # 宿主机path
      path: /data
   # 可选配置
      type: Directory
```

4. nfs 的用法

- 安装 NFS 服务

```
# yum 安装 NFS 服务
yum install -y nfs-utils
# 创建共享目录
mkdir -p /opt/nfs/data
# 编写 NFS 的共享配置
vi /etc/exports
# 输入以下内容 * 代表对所有IP都开发此目录,rw是读写
/opt/nfs/data  *(rw,no_root_squash)
# 启动服务
systemctl enable nfs && systemctl start nfs
# 查看 NFS 共享目录
showmount -e <宿主机IP>
# 检查
ps -ef | grep nfs
```

- Pod 挂载 NFS

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: nginx-dep1
spec:
  replicas: 1
  selector:
    matchLabels:
    - app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        volumeMounts:
        - name: wwwroot
          mountPath: /usr/share/nginx/html
        ports:
        - containerPort: 80
      volumes:
       - name: wwwroot
         nfs:
           server: 10.0.0.103
           path: /opt/nfs/data
```

```
kubectl apply -f nfs.yaml
# 查看pod
kubectl get pods
# 进入pod
kubectl exec -it nginx-dep1-c9cddc7d4-nzcdt bash
# 查看目录
ls /usr/share/nginx/html
```

```
# 在 nfs 服务器上创建 index.html
cd /data/nfs
vim index.html
# 输入内容
hello nfs
```

```
# 在 pod 中再次查看目录，发现已经有index.html 文件
ls /usr/share/nginx/html
```

对外暴露端口

```
kubectl expose deployment nginx-dep1 --port=80 --target-port=80 --type=NodePort
# 查看 svc
kubectl get svc
# 浏览器访问IP 端口
http://IP:port
```

8、PV(PersistentVolume) & PVC(PersistentVolumeClaim)

1. PersistentVolume（PV）

PV 是Volume之类的卷插件，但具有独立于使用 PV 的 Pod 的生命周期，也是集群中的资源
持久化存储，对存储资源镜像抽象，对外提供可以调用的地方（生产者）

- PV 访问模式
  - ReadWriteOnce 该卷可以被单个节点以读/写模式挂载，简写 RWO
  - ReadOnlyMany 该卷可以被多个节点以只读模式挂载，简写 ROX
  - ReadWriteMany 该卷可以被多个节点以读/写模式挂载，简写 RWX
- 回收策略
  - Retain（保留） - 手动回收
  - Recycle（回收） - 基本擦除（rm -rf /thevolume/*）
  - Delete（删除） - 关联的存储资源将被删除
    当前只有 NFS 和 hostPath 支持回收策略，AWS EBS、GCE PD、Azure Disk和 Cinder卷支持删除策略
- PV 状态
  - Available（可用） - 一块空闲资源还没有被任何声明绑定
  - Bound（已绑定） - 卷已被声明绑定
  - Released（已释放） - 声明被删除，但资源还未被集群重新声明
  - Failed（失败） - 该卷自动回收失败
- 使用流程
  - 定义 PV
  - 存储容量 resources
  - 匹配模式 accessModes
  - 定义 PVC - 绑定 PV
- 定义 PV

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfsv1
spec:
  capacity:
    storage: 1Gi
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: nfs
  nfs:
   path: /opt/nfs/data
   server: 10.0.0.101
```

- 执行查看

```
kubectl apply -f pv.yaml
kubectl get pv
```

2. PersistentVolumeClaim（PVC）

PVC 消耗 PV 资源，声明可以请求特点的大小和访问模式（读/写一次 或 只读多次）
用于调用，不需要关心内部实现细节（消费者）

- 定义 PVC

```
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: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15
        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
```

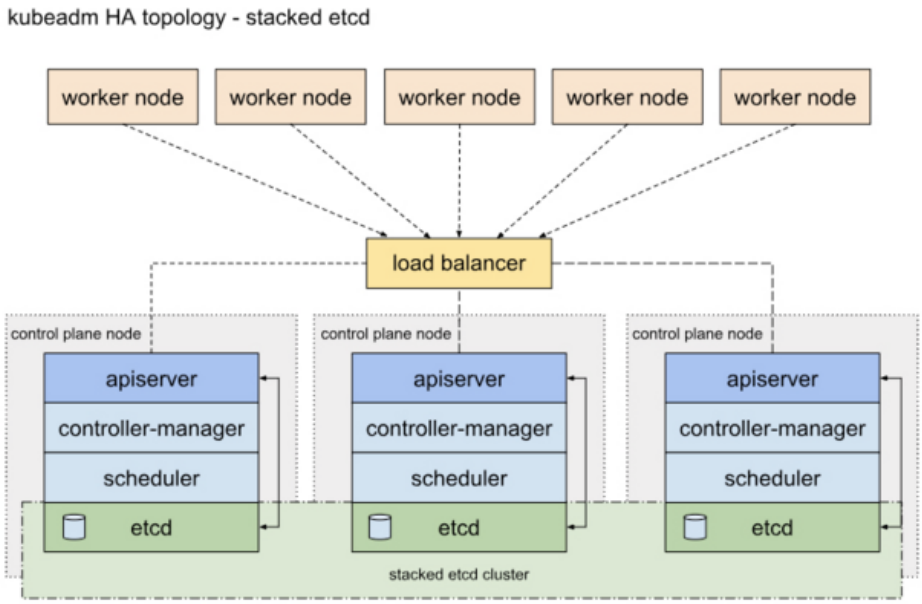
- 执行并查看

```
kubectl apply -f pvc.yaml
kubectl get pvc
```

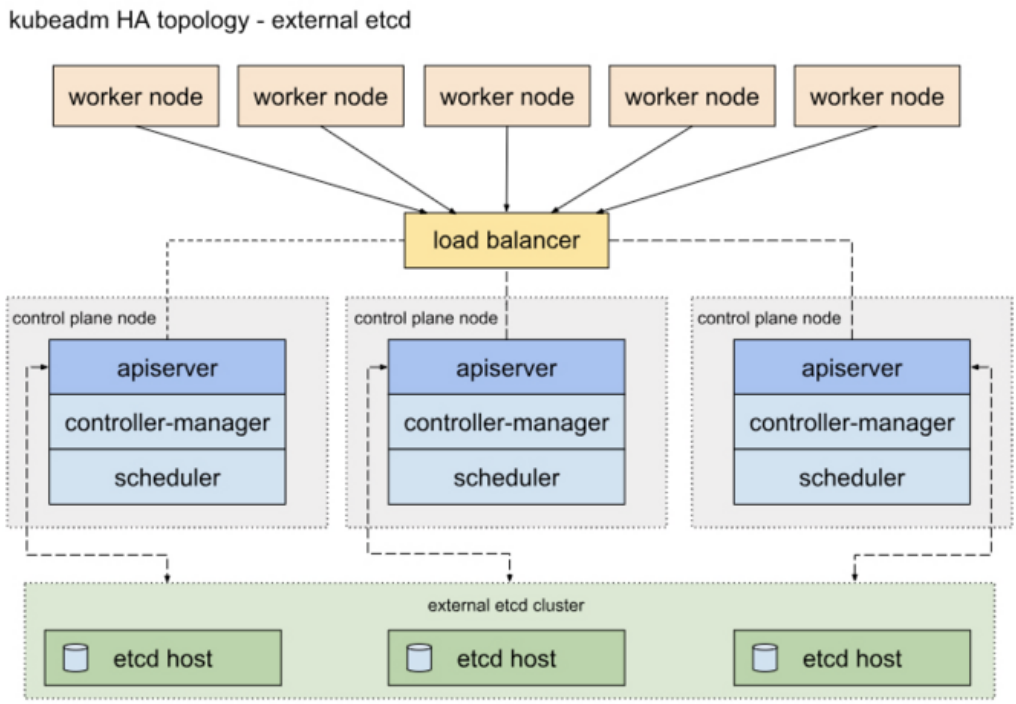
四、集群搭建

1、HA的2种部署方式

- etcd与Master节点组件混布



- 独立的Etcd集群，不与Master节点混布



2、kubeadm 搭建第一种方式的HA集群

1. kubeadm 部署命令

```
# 创建一个 master 节点
kubeadm init
# 将一个 node 节点加入到当前集群
kubeadm join <master节点的IP和端口>
```

2. 机器要求

- 一台或多台机器，操作系统 Centos7.x-86_x64
- 硬件配置：2GB或者更多 RAM，2个CPU或更多，磁盘30GB或更多
- 网络访问
- 禁用 swap 分区

3. 主机规划

角色	IP
master1	10.0.0.101
master2	10.0.0.102
node2	10.0.0.103
VIP（虚拟IP）	10.0.0.100

4. 搭建过程

1.主机优化

```
# 所有节点执行
# 关闭防火墙
systemctl stop firewalld
systemctl disable firewalld
# 关闭selinux
sed -i 's/enforcing/disabled' /etc/selinux/config # 永久关闭
setenforce 0 # 临时
# 关闭swap分区
swapoff -a # 临时
sed -ri 's/.*swap.*/#&/' /etc/fstab # 永久
# 主机名修改
# master1 主机执行
hostnamectl set-hostname master1
# master2 主机执行
hostnamectl set-hostname master2
# node1 主机执行
hostnamectl set-hostname node1
# 允许 iptables 检查桥接流量
cat > /etc/sysctl.d/k8s.conf << EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
sysctl --system # 生效
# 时间同步 - 很重要，不然 etcd 启动有问题
yum install ntpdate -y
ntpdate time.windows.com
```

```
# master节点执行
cat >> /etc/hosts << EOF
10.0.0.100 master.vip.io k8s-vip
10.0.0.101 master01.io  k8s-m1
10.0.0.102 master02.io  k8s-m2
10.0.0.103 node01.io   k8s-n1
EOF
```

2. keepalived 安装[master节点安装]

```
yum install -y conntrack-tools libseccomp libtool-ltdl
yum install -y keepalived
```

配置 master 节点 1

```
cat > /etc/keepalived/keepalived.conf <<EOF
! Configuration File for keepalived
global_defs {
  router_id k8s
}

vrrp_script check_haproxy {
  script "killall -0 haproxy"
  interval 3
  weight -2
  fall 10
  rise 2
}

vrrp_instance VI_1 {
  state MASTER  # MASTER
  interface ens33  # 本机网卡名
  virtual_router_id 51
  priority 250 # 权重
  advert_int 1
  authentication {
     auth_type PASS
     auth_pass ceb1b3ec013d66163d6ab
  }
  virtual_ipaddress {
     10.0.0.100 # 虚拟IP
  }
  track_script {
     check_haproxy # 模块
  }
}
EOF
```

配置 master 节点 2 [state,priority 和 master 1 不一样]

```
cat > /etc/keepalived/keepalived.conf <<EOF
! Configuration File for keepalived

global_defs {
  router_id k8s
}

vrrp_script check_haproxy {
  script "killall -0 haproxy"
  interval 3
  weight -2
  fall 10
  rise 2
}

vrrp_instance VI_1 {
  state BACKUP  # BACKUP
  interface ens33   # 本机网卡名
  virtual_router_id 51
  priority 200    # 权重
```

```
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass ceb1b3ec013d66163d6ab
    }
    virtual_ipaddress {
        10.0.0.100   # 虚拟IP
    }
    track_script {
        check_haproxy  # 模块
    }
}
EOF
```

启动、检查 keepalived

```
# 启动
systemctl start keepalived
# 开机自启动
systemctl enable keepalived
# 查看启动状态
systemctl status keepalived
# 查看网卡信息 - 查看虚拟 IP
ip a s ens33
```

3. haproxy 安装[master 节点安装]

```
yum install -y haproxy
```

两台master节点的配置均相同，配置中声明了后端代理的两个master节点服务器，指定了haproxy运行的端口为16443等，因此16443端口为集群的入口

```
cat > /etc/haproxy/haproxy.cfg << EOF
#---------------------------------------------------------------------
# Global settings
#---------------------------------------------------------------------
global
    # to have these messages end up in /var/log/haproxy.log you will
    # need to:
    # 1) configure syslog to accept network log events.  This is done
    #    by adding the '-r' option to the SYSLOGD_OPTIONS in
    #    /etc/sysconfig/syslog
    # 2) configure local2 events to go to the /var/log/haproxy.log
    #   file. A line like the following can be added to
    #   /etc/sysconfig/syslog
    #
    #    local2.*                       /var/log/haproxy.log
    #
    log         127.0.0.1 local2
    chroot      /var/lib/haproxy
    pidfile     /var/run/haproxy.pid
    maxconn     4000
    user        haproxy
    group       haproxy
    daemon
    # turn on stats unix socket
```

```
  stats socket /var/lib/haproxy/stats
#---------------------------------------------------------------------
# common defaults that all the 'listen' and 'backend' sections will
# use if not designated in their block
#---------------------------------------------------------------------

defaults
  mode          http
  log           global
  option         httplog
  option         dontlognull
  option http-server-close
  option forwardfor    except 127.0.0.0/8
  option         redispatch
  retries         3
  timeout http-request  10s
  timeout queue      1m
  timeout connect     10s
  timeout client     1m
  timeout server     1m
  timeout http-keep-alive 10s
  timeout check      10s
  maxconn         3000
#---------------------------------------------------------------------
# kubernetes apiserver frontend which proxys to the backends
#---------------------------------------------------------------------

frontend kubernetes-apiserver
  mode         tcp
  bind         *:16443
  option        tcplog
  default_backend   kubernetes-apiserver
#---------------------------------------------------------------------
# round robin balancing between the various backends
#---------------------------------------------------------------------

backend kubernetes-apiserver
  mode    tcp
  balance   roundrobin
  server   master01.io  10.0.0.101:6443 check
  server   master02.io  10.0.0.102:6443 check
#---------------------------------------------------------------------
# collection haproxy statistics message
#---------------------------------------------------------------------

listen stats
  bind         *:1080
  stats auth      admin:awesomePassword
  stats refresh    5s
  stats realm     HAProxy Statistics
  stats uri      /admin?stats

EOF
```

启动、检查 haproxy

```
# 启动
systemctl start haproxy
# 开机自启动
systemctl enable haproxy
# 查看启动状态
systemctl status haproxy
# 端口检查
netstat -lntup | grep haroxy
```

4. 所有节点安装 Docker、kubeadm、kubelet

- 安装 Docker

```
# 移除旧的 Docker环境
sudo yum remove docker
            docker-client
            docker-client-latest
            docker-common
            docker-latest
            docker-latest-logrotate
            docker-logrotate
            docker-engine
# 安装依赖
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo
https://download.docker.com/linux/centos/docker-ce.repo
# 安装
sudo yum install docker-ce docker-ce-cli containerd.io
# 启动 Docker
sudo systemctl start docker
# 开机自启动
sudo systemctl enable docker
# 查看 Docker
sudo docker --version
```

- Docker 镜像加速 - 阿里云镜像加速

```
cat > /etc/docker/daemon.json << EOF
{
    "registry-mirrors": ["https://b9pmyelo.mirror.aliyuncs.com"]
}
EOF
```

- kubernetes 阿里云 yum 源

```
cat > /etc/yum.repos.d/kubernetes.repo << EOF
[kubernetes]
name=Kubernetes
baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=0
repo_gpgcheck=0
gpgkey=https://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg
https://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg
EOF
```

- 安装 kubeadm、kubelet、kubectl

```
yum install -y kubelet-1.16.3 kubeadm-1.16.3 kubectl-1.16.3
# 开机自启动
systemctl enable kubelet
```

5. 创建 kubeadm 配置文件并在VIP主机上执行

> 在具体VIP的master节点上操作，haproxy 配置的 priority master1的较大，此时VIP在master1节点上

```
# 创建目录
mkdir /usr/local/kubernetes/manifests -p
# 进入目录
cd /usr/local/kubernetes/manifests/
# 创建 yaml
vi kubeadm-config.yaml
```

硅谷教程：输入以下内容

```
apiServer:
  certSANs:
    - k8s-m1
    - k8s-m2
    - master.vip.io
    - 10.0.0.101
    - 10.0.0.102
    - 10.0.0.100
    - 127.0.0.1
  extraArgs:
    authorization-mode: Node,RBAC
  timeoutForControlPlane: 4m0s
apiVersion: kubeadm.k8s.io/v1beta1
certificatesDir: /etc/kubernetes/pki
clusterName: kubernetes
controlPlaneEndpoint: "master.vip.io:16443"
controllerManager: {}
dns:
  type: CoreDNS
etcd:
  local:
    dataDir: /var/lib/etcd
imageRepository: registry.aliyuncs.com/google_containers
kind: ClusterConfiguration
kubernetesVersion: v1.16.3
networking:
  dnsDomain: cluster.local
  podSubnet: 10.244.0.0/16
  serviceSubnet: 10.1.0.0/16
scheduler: {}
```

官方教程

```
kubeadm config print init-defaults > kubeadm-config.yaml
# 修改初始化配置文件 kubeadm-config.yaml 如下注意点
cat kubeadm-config.yaml
```

```
apiVersion: kubeadm.k8s.io/v1beta2
bootstrapTokens:
- groups:
 - system:bootstrappers:kubeadm:default-node-token
 token: abcdef.0123456789abcdef
 ttl: 24h0m0s
 usages:
 - signing
 - authentication
kind: InitConfiguration
localAPIEndpoint:
 advertiseAddress: 10.0.0.101   # 本机IP
 bindPort: 6443
nodeRegistration:
 criSocket: /var/run/dockershim.sock
 name: master1    # 本主机名
 taints:
 - effect: NoSchedule
  key: node-role.kubernetes.io/master
---
apiServer:
 timeoutForControlPlane: 4m0s
apiVersion: kubeadm.k8s.io/v1beta2
certificatesDir: /etc/kubernetes/pki
clusterName: kubernetes
controlPlaneEndpoint: "10.0.0.100:16443"  # 虚拟IP和haproxy端口
controllerManager: {}
dns:
 type: CoreDNS
etcd:
 local:
  dataDir: /var/lib/etcd
imageRepository: registry.aliyuncs.com/google_containers # 阿里云镜像仓库源
kind: ClusterConfiguration
kubernetesVersion: v1.16.3   # k8s版本
networking:
 dnsDomain: cluster.local
 podSubnet: "10.244.0.0/16"
 serviceSubnet: 10.96.0.0/12
scheduler: {}
---
apiVersion: kubeproxy.config.k8s.io/v1alpha1
kind: KubeProxyConfiguration
featureGates:
 SupportIPVSProxyMode: true
mode: ipvs
```

集群初始化

```
# 执行
kubeadm init --config kubuadm-config.yaml
```

若初始化失败重置集群

```
kubeadm reset
```

按照提示操作

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
# 查看节点信息
kubectl get nodes
kubectl get pods -n kube-system
# 查看集群状态
kubectl get cs
```

安装提示保存好注册信息(token有效期24h)

```
kubeadm join master.vip.io:16443 --token jv5z7n.3y1zi95p952y9p65
  --discovery-token-ca-cert-hash
sha256:403bca185c2f3a4791685013499e7ce58f9848e2213e27194b75a2e3293d8812
  --control-plane
```

注册信息若忘记或过期，重新生成

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

安装集群网络

```
mkdir flannel
cd flannel
# 下载 flannel.yaml
wget -c
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-
flannel.yml
# 安装
kubectl apply -f kube-flannel.yaml
# 检查
kubectl get pods -n kube-system
```

6. master2 节点加入集群

- 复制秘钥及相关文件
  从 master1 复制秘钥及相关文件到 master2

```
ssh root@10.0.0.102 mkdir -p /etc/kubernetes/pki/etcd
scp /etc/kubernetes/admin.conf root@10.0.0.102:/etc/kubernetes
scp /etc/kubernetes/pki/{ca.*,sa.*,front-proxy-ca.*}
root@10.0.0.102:/etc/kubernetes/pki
scp /etc/kubernetes/pki/etcd/ca.* root@10.0.0.102:/etc/kubernetes/pki/etcd
```

- master2加入集群，需要带上 --control-plane (master1上init输出的join命令)

```
kubeadm join master.vip.io:16443 --token jv5z7n.3y1zi95p952y9p65 --discovery-token-ca-cert-hash
sha256:403bca185c2f3a4791685013499e7ce58f9848e2213e27194b75a2e3293d8812 --control-plane
```

- 检查状态

```
kubectl get node
kubectl get pods --all-namespaces
```

7. node1 节点加入集群

- 复制秘钥及相关文件
  从 master1 复制 admin.conf 文件到 node1

```
scp /etc/kubernetes/admin.conf root@10.0.0.103:/etc/kubernetes
```

- node1加入集群（master1上init输出的join命令）

```
kubeadm join master.vip.io:16443 --token jv5z7n.3y1zi95p952y9p65 --discovery-token-ca-cert-hash
sha256:403bca185c2f3a4791685013499e7ce58f9848e2213e27194b75a2e3293d8812
```

- master1 集群网络重新安装(master1主机上重新执行)

```
kubectl apply -f kube-flannel.yaml
```

- 检查状态

```
kubectl get node
kubectl get pods --all-namespaces
```

8. 测试kubernetes集群

```
kubectl create deployment nginx --image=nginx
kubectl expose deployment nginx --port=80 --type=NodePort
kubectl get pod,svc
```

3、集群安全机制 RBAC

1. 资源角色

- Role 角色
- ClusterRole 集群角色
- RoleBinding 角色绑定
- ClusterRoleBinding 集群角色绑定

2. 认证、鉴权、访问控制

- 认证
- 使用 Bootstrap 令牌进行身份认证
- 准入控制器
- 动态准入控制

- 鉴权
- 基于角色的访问控制
- 基于属性的访问控制
- 节点鉴权
- Webhook 鉴权
- 证书签名请求
- 包括 CSR 认证 和证书签名
- 服务账户
- 开发者指导
- 管理

五、集群监控

1、监控指标

1. 集群监控

- 节点资源利用率
- 节点树
- 运行 pods

2. Pod 监控

- 容器指标
- 应用程序

2、监控平台

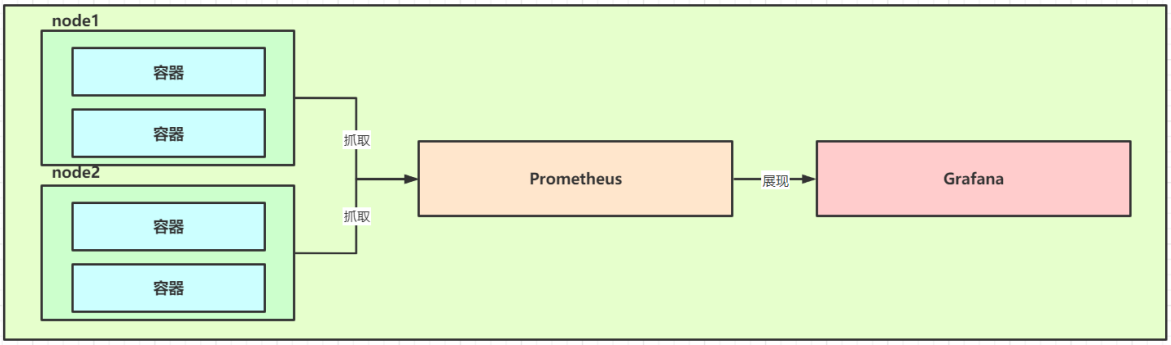
1. Prometheus

- 开源的
- 监控、告警、数据库
- 以 HTTP 协议周期性抓取被监控组件状态
- 不需要复杂的集成过程，使用http接口接入

2. Grafana

- 开源的，数据分析和可视化工具
- 支持多种数据源

3、处理流程图



4、监控平台搭建

1. 部署 Prometheus

- node-exporter.yaml

```yaml
---
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: node-exporter
  namespace: kube-system
  labels:
    k8s-app: node-exporter
spec:
  template:
    metadata:
      labels:
        k8s-app: node-exporter
    spec:
      containers:
      - image: prom/node-exporter
        name: node-exporter
        ports:
        - containerPort: 9100
          protocol: TCP
          name: http
---
apiVersion: v1
kind: Service
metadata:
  labels:
    k8s-app: node-exporter
  name: node-exporter
  namespace: kube-system
spec:
  ports:
  - name: http
    port: 9100
    nodePort: 31672
    protocol: TCP
  type: NodePort
  selector:
    k8s-app: node-exporter
```

- rbac-setup.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups: [""]
  resources:
  - nodes
  - nodes/proxy
  - services
  - endpoints
  - pods
  verbs: ["get", "list", "watch"]
- apiGroups:
```

```
  - extensions
  resources:
  - ingresses
  verbs: ["get", "list", "watch"]
- nonResourceURLs: ["/metrics"]
  verbs: ["get"]
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
- kind: ServiceAccount
  name: prometheus
  namespace: kube-system
```

- configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: kube-system
data:
  prometheus.yml: |
    global:
      scrape_interval:     15s
      evaluation_interval: 15s
    scrape_configs:

    - job_name: 'kubernetes-apiservers'
      kubernetes_sd_configs:
      - role: endpoints
      scheme: https
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      relabel_configs:
      - source_labels: [__meta_kubernetes_namespace,
__meta_kubernetes_service_name, __meta_kubernetes_endpoint_port_name]
        action: keep
        regex: default;kubernetes;https

    - job_name: 'kubernetes-nodes'
      kubernetes_sd_configs:
      - role: node
      scheme: https
```

```
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      relabel_configs:
      - action: labelmap
        regex: __meta_kubernetes_node_label_(.+)
      - target_label: __address__
        replacement: kubernetes.default.svc:443
      - source_labels: [__meta_kubernetes_node_name]
        regex: (.+)
        target_label: __metrics_path__
        replacement: /api/v1/nodes/${1}/proxy/metrics

    - job_name: 'kubernetes-cadvisor'
      kubernetes_sd_configs:
      - role: node
      scheme: https
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      relabel_configs:
      - action: labelmap
        regex: __meta_kubernetes_node_label_(.+)
      - target_label: __address__
        replacement: kubernetes.default.svc:443
      - source_labels: [__meta_kubernetes_node_name]
        regex: (.+)
        target_label: __metrics_path__
        replacement: /api/v1/nodes/${1}/proxy/metrics/cadvisor

    - job_name: 'kubernetes-service-endpoints'
      kubernetes_sd_configs:
      - role: endpoints
      relabel_configs:
      - source_labels:
[__meta_kubernetes_service_annotation_prometheus_io_scrape]
        action: keep
        regex: true
      - source_labels:
[__meta_kubernetes_service_annotation_prometheus_io_scheme]
        action: replace
        target_label: __scheme__
        regex: (https?)
      - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_path]
        action: replace
        target_label: __metrics_path__
        regex: (.+)
      - source_labels: [__address__,
__meta_kubernetes_service_annotation_prometheus_io_port]
        action: replace
        target_label: __address__
        regex: ([^:]+)(?::\d+)?;(\d+)
        replacement: $1:$2
      - action: labelmap
        regex: __meta_kubernetes_service_label_(.+)
      - source_labels: [__meta_kubernetes_namespace]
        action: replace
        target_label: kubernetes_namespace
```

```
      - source_labels: [__meta_kubernetes_service_name]
        action: replace
        target_label: kubernetes_name

    - job_name: 'kubernetes-services'
      kubernetes_sd_configs:
      - role: service
      metrics_path: /probe
      params:
        module: [http_2xx]
      relabel_configs:
      - source_labels:
[__meta_kubernetes_service_annotation_prometheus_io_probe]
        action: keep
        regex: true
      - source_labels: [__address__]
        target_label: __param_target
      - target_label: __address__
        replacement: blackbox-exporter.example.com:9115
      - source_labels: [__param_target]
        target_label: instance
      - action: labelmap
        regex: __meta_kubernetes_service_label_(.+)
      - source_labels: [__meta_kubernetes_namespace]
        target_label: kubernetes_namespace
      - source_labels: [__meta_kubernetes_service_name]
        target_label: kubernetes_name

    - job_name: 'kubernetes-ingresses'
      kubernetes_sd_configs:
      - role: ingress
      relabel_configs:
      - source_labels:
[__meta_kubernetes_ingress_annotation_prometheus_io_probe]
        action: keep
        regex: true
      - source_labels:
[__meta_kubernetes_ingress_scheme,__address__,__meta_kubernetes_ingress_path]
        regex: (.+);(.+);(.+)
        replacement: ${1}://${2}${3}
        target_label: __param_target
      - target_label: __address__
        replacement: blackbox-exporter.example.com:9115
      - source_labels: [__param_target]
        target_label: instance
      - action: labelmap
        regex: __meta_kubernetes_ingress_label_(.+)
      - source_labels: [__meta_kubernetes_namespace]
        target_label: kubernetes_namespace
      - source_labels: [__meta_kubernetes_ingress_name]
        target_label: kubernetes_name

    - job_name: 'kubernetes-pods'
      kubernetes_sd_configs:
      - role: pod
      relabel_configs:
      - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
        action: keep
```

```
        regex: true
      - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_path]
        action: replace
        target_label: __metrics_path__
        regex: (.+)
      - source_labels: [__address__,
__meta_kubernetes_pod_annotation_prometheus_io_port]
        action: replace
        regex: ([^:]+)(?::\d+)?;(\d+)
        replacement: $1:$2
        target_label: __address__
      - action: labelmap
        regex: __meta_kubernetes_pod_label_(.+)
      - source_labels: [__meta_kubernetes_namespace]
        action: replace
        target_label: kubernetes_namespace
      - source_labels: [__meta_kubernetes_pod_name]
        action: replace
        target_label: kubernetes_pod_name
```

- prometheus.deploy.yml

```yaml
---
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  labels:
    name: prometheus-deployment
  name: prometheus
  namespace: kube-system
spec:
  replicas: 1
  selector:
    matchLabels:
      app: prometheus
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      containers:
      - image: prom/prometheus:v2.0.0
        name: prometheus
        command:
        - "/bin/prometheus"
        args:
        - "--config.file=/etc/prometheus/prometheus.yml"
        - "--storage.tsdb.path=/prometheus"
        - "--storage.tsdb.retention=24h"
        ports:
        - containerPort: 9090
          protocol: TCP
        volumeMounts:
        - mountPath: "/prometheus"
          name: data
        - mountPath: "/etc/prometheus"
          name: config-volume
```

```
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
          limits:
            cpu: 500m
            memory: 2500Mi
      serviceAccountName: prometheus
      volumes:
      - name: data
        emptyDir: {}
      - name: config-volume
        configMap:
          name: prometheus-config
```

- prometheus.svc.yml

```
---
kind: Service
apiVersion: v1
metadata:
  labels:
    app: prometheus
  name: prometheus
  namespace: kube-system
spec:
  type: NodePort
  ports:
  - port: 9090
    targetPort: 9090
    nodePort: 30003
  selector:
    app: prometheus
```

```
# 部署
kubectl apply -f .
# 查看
kubectl get pods -n kube-system
```

2. 部署 Grafana

- grafana-deploy.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: grafana-core
  namespace: kube-system
  labels:
    app: grafana
    component: core
spec:
  replicas: 1
  template:
    metadata:
      labels:
```

```
        app: grafana
        component: core
    spec:
      containers:
      - image: grafana/grafana:4.2.0
        name: grafana-core
        imagePullPolicy: IfNotPresent
        # env:
        resources:
          # keep request = limit to keep this container in guaranteed class
          limits:
            cpu: 100m
            memory: 100Mi
          requests:
            cpu: 100m
            memory: 100Mi
        env:
          # The following env variables set up basic auth twith the default
admin user and admin password.
          - name: GF_AUTH_BASIC_ENABLED
            value: "true"
          - name: GF_AUTH_ANONYMOUS_ENABLED
            value: "false"
          # - name: GF_AUTH_ANONYMOUS_ORG_ROLE
          #   value: Admin
          # does not really work, because of template variables in exported
dashboards:
          # - name: GF_DASHBOARDS_JSON_ENABLED
          #   value: "true"
        readinessProbe:
          httpGet:
            path: /login
            port: 3000
          # initialDelaySeconds: 30
          # timeoutSeconds: 1
        volumeMounts:
        - name: grafana-persistent-storage
          mountPath: /var
      volumes:
      - name: grafana-persistent-storage
        emptyDir: {}
```

- grafana-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: grafana
  namespace: kube-system
  labels:
    app: grafana
    component: core
spec:
  type: NodePort
  ports:
    - port: 3000
  selector:
```

```
    app: grafana
    component: core
```

- grafana-ing.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
   name: grafana
   namespace: kube-system
spec:
   rules:
   - host: k8s.grafana
     http:
       paths:
       - path: /
         backend:
          serviceName: grafana
          servicePort: 3000
```

```
# 部署
kubectl apply -f .
# 查看
kubectl get pods -n kube-system
```

3. 打开 Grafana ，配置数据源，设置显示模板

- 访问服务

```
kubectl get svc
# 记住 prometheus 的 内部CLUSTER_IP 和端口默认9090
# 用于配置 grafana 的 数据源
# 访问 grafana 的对外暴露端口，访问服务
```

- 登录

```
admin/admin
```

- 添加数据源

```
# 选择 prometheus
# 注意 ip 是 ClusterIP
```

- 设置显示模板

```
import -> 315 -> db
```

六、Helm

1、What

Helm 是官方提供的类似于 yum 的包管理工具，可以很方便的将打包好的 yaml 部署到 kubernetes上

2、Why

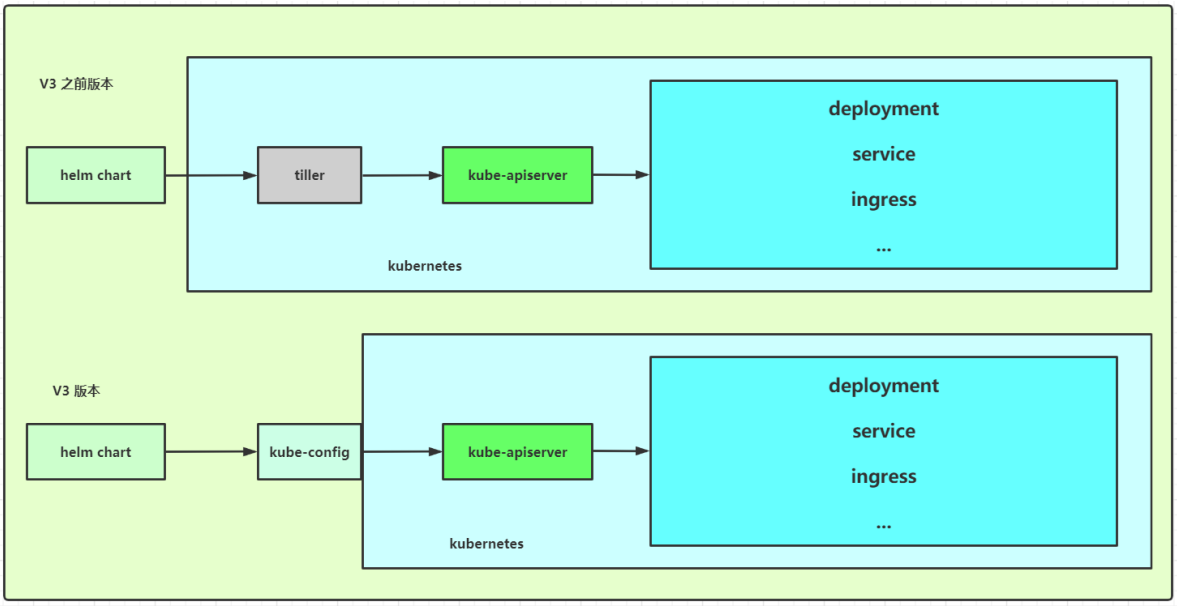
- 诸多的 yaml 文件需要管理维护
- 如何将诸多的服务作为一个整体管理
- 如何高效复用诸多的资源文件
- 如何支持应用级别的版本管理

3、How

1. Helm 3个重要概念

- helm
  一个命令行客户端工具，主要用于 Kubernetes 应用 chart 的创建、打包、发布和管理
- Chart
  应用描述，一系列用于庙宇 k8s 资源相关文件的集合
- Release
  基于 Chart 的部署实体，一个 chart 被 helm 运行后将会生成对应的一个 release，将在 k8s 中创建出真实运行的资源对象

2. Helm v3 版本改版前后的变化



- 删除 Tiller
- Release 名称可以在不同命名空间重用
- 支持将 Chart 推送至 Docker 镜像仓库中
- 使用 JSONSchema 验证 chart values

3. Helm 客户端

- 部署 helm 客户端
  下载 helm 客户端

```
# 下载
wget https://get.helm.sh/helm-vv3.2.1-linux-amd64.tar.gz
# 解压
tar -zxvf helm-vv3.2.1-linux-amd64.tar.gz
# 移动 helm 命令到 /usr/bin 目录下
mv linux-amd64/helm /usr/bin/
```

- 常用命令

命令	描述
create	创建一个 chart 并指定名称
dependency	管理 chart 依赖
get	下载一个 release，可用子命令：all、hooks、manifest、notes、values
history	获取 release 历史
install	安装一个 chart
list	列出 release
package	将 chart 目录打包到 chart 存档文件中
pull	从远程仓库中下载 chart 并解压到本地 helm pull stable/mysql --untar
repo	统计，列出，移除，更新和索引 chart 仓库，可用子命令：add、index、list、remove、update
rollback	从之前版本回滚
search	根据关键字搜索chart，可用子命令：hub、repo
show	查看 chart 详细信息，可用子命令：all、chart、readme、values
status	显示已命名版本的状态
template	本地呈现模板
uninstall	卸载一个 release
upgrade	更新一个 release
version	查看 helm 客户端版本

- chart 仓库
- 微软仓库

```
http://mirror.azure.cn/kubernetes/charts/
```

- 阿里云仓库

```
https://kubernetes.oss-cn-hangzhou.aliyuncs.com/charts
```

- 官方仓库

```
https://hub.kubeapps.com/charts/incubator
```

- 仓库操作

```
# 添加仓库
helm repo add stable http://mirror.azure.cn/kubernetes/charts/
helm repo add aliyun https://kubernetes.oss-cn-hangzhou.aliyuncs.com/charts
helm repo update
# 查看配置的仓库
helm repo list
helm search repo stable
# 删除储存库
helm repo remove aliyun
```

4. helm 基本使用

- 主要三个命令
  - chart install
  - chart upgrade
  - chart rollback
- 使用 chart 部署一个应用

```
# 查找 chart
helm search repo weave
# 查看 chart 信息
helm show chart stable/mysql
# 安装包
helm install ui stable/weave-scope
# 查看发布状态
helm list
helm status ui
# 修改service type:NodePort 即可访问 ui
helm upgrade ui --set service.type=NodePort
```

- helm install 命令可以从多个来源安装
  - chart 储存库
  - 本地 chart 存档（helm install foo-0.0.1.tgz）
  - chart 目录（helm install /path/foo）
  - 完整的 URL （helm install https://example.com/charts/foo-0.0.1.tgz）
- 构建一个 helm chart

```
# 创建 chart
helm create mychart
# 查看目录结构
tree mychart
```

  - Chart.yaml
    用于描述这个 Chart 的基本信息，包括名字、描述信息以及版本
  - values.yaml
    用于存储 templates 目录中目标文件中使用到的变量的值
  - templates
    此目录里存放所有 yaml 模板文件
  - charts
    此目录存放这个 chart 依赖的所有子chart
  - NOTES.txt
    介绍 chart 帮助信息

    - _helpers.tpl
      放置模板助手的地方，可以在整个 chart 中重复使用
- 部署自定义 chart

```
# 部署
helm install web mychart/
# 打包
helm package mychart/
```

- helm 安装 release 两种传递配置方式
    - --values | -f
      使用 yaml 文件覆盖配置，可以指定多次，优先使用最右边的文件
    - --set
      通过命令行的方式对指定项进行覆盖
- yaml 和 --set 的关系

yaml	**set**
name: value	--set name=value
a:b c:d	--set a=b,c=d
outer: inner: value	--set outer.inner=value
name: - a - b - c	--set name={a,b,c}
servers: - port: 80	--set servers[0].port=80

yaml	set
servers: - port: 80 - host: example	--set servers[0].port=80,servers[0].host=example
name: "value1,value2"	--set name=value1,value2
nodeSelector: kubernetes.ui/role: master	--set nodeSelector."kubernetes.io/role"=master

- values.yaml 配置模板
  通过表达式形式使用全局变量 **{{.Values.变量名称}}** 或 **{{.Release.Name}} (动态生成名称)**

```
helm create nginx
# values.yaml 文件配置全局变量的值
vim nginx/values.yaml
# 输入以下内容
replicas: 3
image: nginx
tag: 1.15
serviceport: 80
targetport: 80
label: nginx
```

```
# templates 目录下的 yaml 资源文件使用全局变量
vim nginx/templates/deployment.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: {{.Values.label}}
# 生成唯一的名称
  name: {{.Release.Name}}
spec:
  replicas: {{.Values.replicas}}
  selector:
    matchLabels:
      app: {{.Values.label}}
  template:
    metadata:
      labels:
        app: {{.Values.label}}
    spec:
      containers:
      - image: {{.Values.image}}:{{.Values.tag}}
        name: web
```

```
 # templates 目录下的 yaml 资源文件使用全局变量
 vim nginx/templates/service.yaml
```

```
apiVersion: v1
kind: Service
```

```
metadata:
  labels:
    app: {{.Values.label}}
  name: {{.Release.Name}}
spec:
  ports:
  - port: {{.Values.serviceport}}
    protocol: TCP
    targetPort: {{.Values.targetport}}
  selector:
    app: {{.Values.label}}
  type: NodePort
```

查看实际的模板被渲染后的资源文件

```
helm get manifest web
helm install web nginx/
helm list
kubectl get pod
```

调试

```
helm install web --dry-run nginx/
```

- 内置对象

内置对象	含义
Release.Name	release 名称
Release.Namespace	release 命名空间
Release.Service	release 服务的名称
Release.Revision	release 修订版本号，从 1 开始累加

- Values 对象值来源
- chart 包中的 values.yaml
- 父 chart 包的 values.yaml
- helm install 或 helm upgrade 的 -f 或 --values 参数传入的自定义 yaml 文件
- -- set 参数传入的值
- 升级、回滚、删除

```
# 升级
helm upgrade --set image.tag=1.17 web nginx
helm upgrade -f values.yaml web nginx
```

```
# 回滚
helm rollback web 1
```

```
# 删除
helm uninstall web
```

```
# 查看历史版本信息
helm get all --revision 1 web
```

- 其他操作参考官方文档

七、java 项目镜像制作

1、编写 java 项目

```
demojenkins
```

2、执行 maven 命令打包

```
mvn clean package
# 得到jar包 demojenkins.jar
```

3、编写 Dockerfile 文件

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ADD ./target/demojenkins.jar demojenkins.jar
ENTRYPOINT ["java","-jar","/demojenkins.jar", "&"]
```

4、制作镜像

```
docker build -t java-demo:v1 .
```

- -t 表示 打 tag
- . 表示Dockerfile文件的位置在当前位置

5、查看镜像

```
docker images
```

6、运行容器

```
docker run -d -p 8111:8111 java-demo:v1 -t
```

7、阿里云镜像服务

- 申请个人版
- 创建命名空间
- 创建镜像仓库
- 访问凭证设置固定密码

8、上传镜像

- 服务器登陆阿里云服务

```
docker login --username=weww**** registry.cn-hangzhou.aliyuncs.com
```

- 镜像 tag

```
docker tag 7c0cbef59328 registry.cn-hangzhou.aliyuncs.com/weww***/java-pro-01:[镜像版本号]
```

- 推送镜像到阿里云服务

```
docker push registry.cn-hangzhou.aliyuncs.com/weww***/java-pro-01:[镜像版本号]
```

- 其他服务器拉去镜像测试

```
# 先登录阿里云服务
docker login --username=weww**** registry.cn-hangzhou.aliyuncs.com
# 拉取镜像
docker pull registry.cn-hangzhou.aliyuncs.com/weww***t/java-pro-01:[镜像版本号]
# 查看镜像
docker images
```

- 启动容器

```
docker run -d --name javademo -p8111:8111 -t
```

- 查看容器启动情况

```
docker ps
```

9、k8s 操作

- deloyment

```
kubectl create deployment javademo1 --image=registry.cn-hangzhou.aliyuncs.com/weww***/java-pro-01:1.0.0 --dry-run -o yaml > javademo.yaml
# 查看文件
cat javademo.yaml
# 应用
kubectl apply -f javademo.yaml
# 查看 pods
kubectl get pods -o wide
# 扩容
kubectl scale deployment javademo1 --replicas=3
```

- service 暴露服务

```
kubectl expose deployment javademo1 --port=8111 --target-port=8111 --type=NodePort
# 查看 svc
kubectl get pods,svc -o wide
```

- 服务访问

```
curl 10.0.0.225:随机端口/user
```