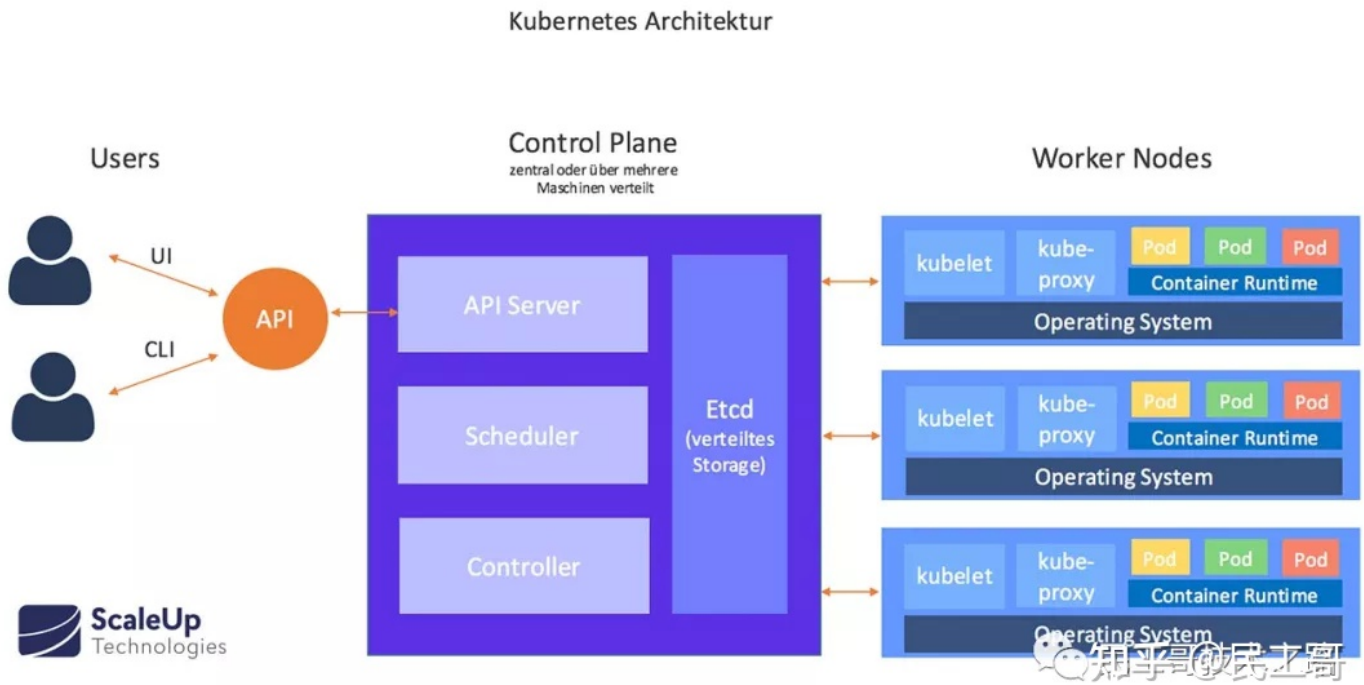
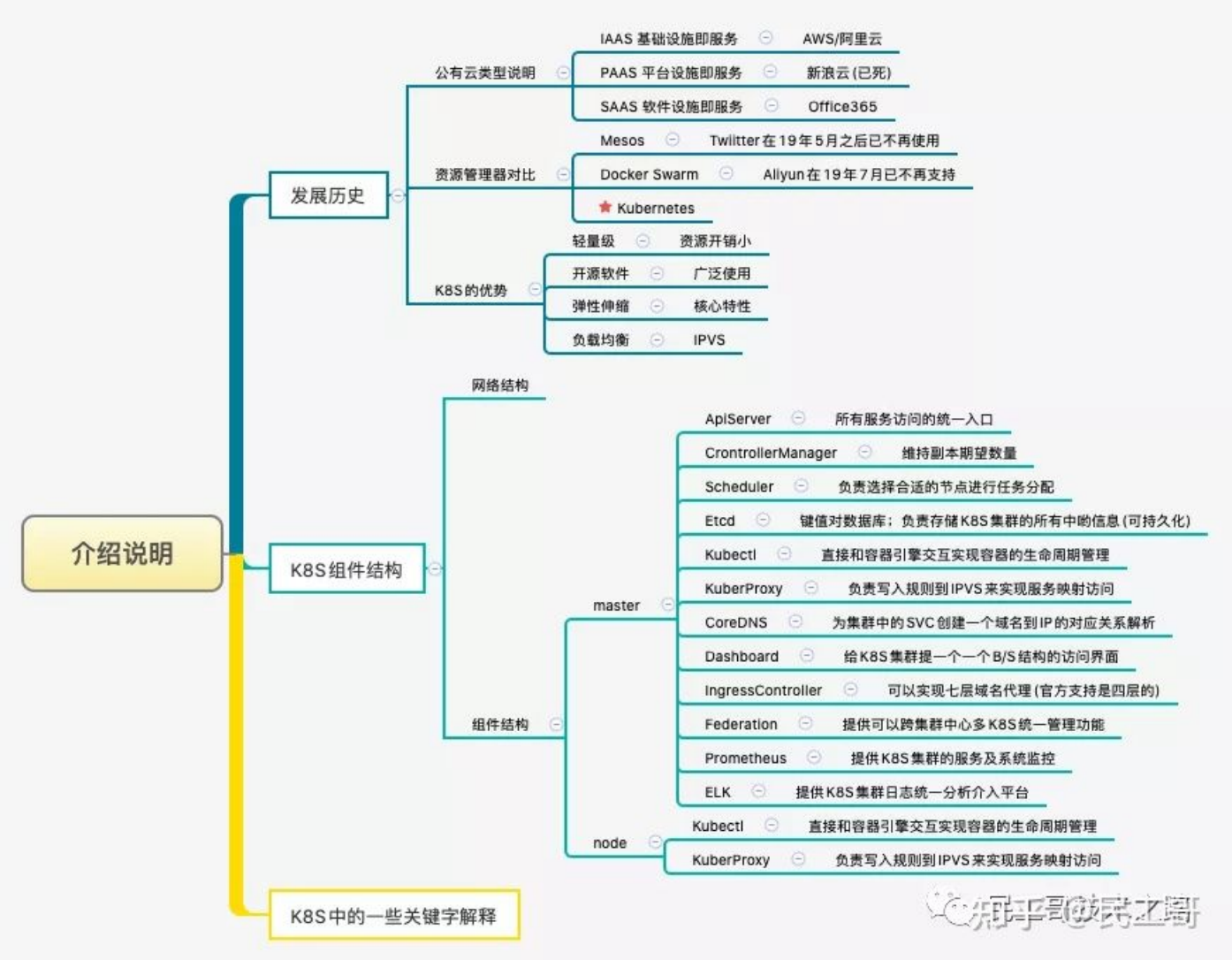
K8S 学习笔记总结



kubernetes 已经成为容器编排领域的王者，它是基于容器的集群编排引擎，具备扩展集群、滚动升级回滚、弹性伸缩、自动治愈、服务发现等多种特性能力。

kubernetes 介绍



Kubernetes 解决的核心问题

- 服务发现和负载均衡
  - Kubernetes 可以使用 DNS 名称或自己的 IP 地址公开容器，如果到容器的流量很大，Kubernetes 可以负载均衡并分配网络流量，从而使部署稳定。

- 存储编排
  - Kubernetes 允许您自动挂载您选择的存储系统，例如本地存储、公共云提供商等。
- 自动部署和回滚
  - 您可以使用 Kubernetes 描述已部署容器的所需状态，它可以以受控的速率将实际状态更改为所需状态。例如，您可以自动化 Kubernetes 来为您的部署创建新容器，删除现有容器并将它们的所有资源用于新容器。

- 自动二进制打包
  - Kubernetes 允许您指定每个容器所需 CPU 和内存（RAM）。当容器指定了资源请求时，Kubernetes 可以做出更好的决策来管理容器的资源。

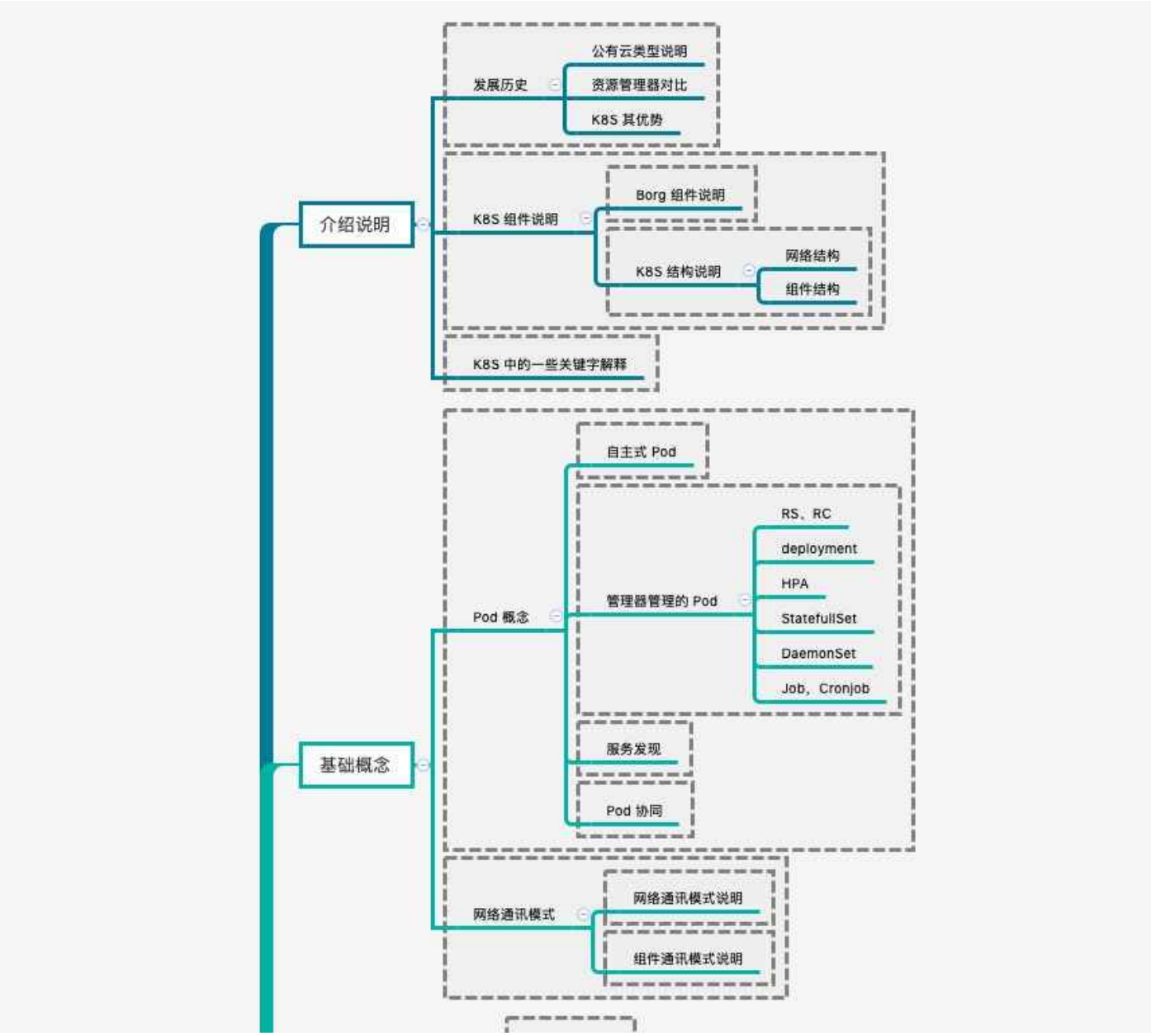
- 自我修复
  - Kubernetes 重新启动失败的容器、替换容器、杀死不响应用户定义的运行状况检查的容器，并且在准备好服务之前不将其通告给客户端。

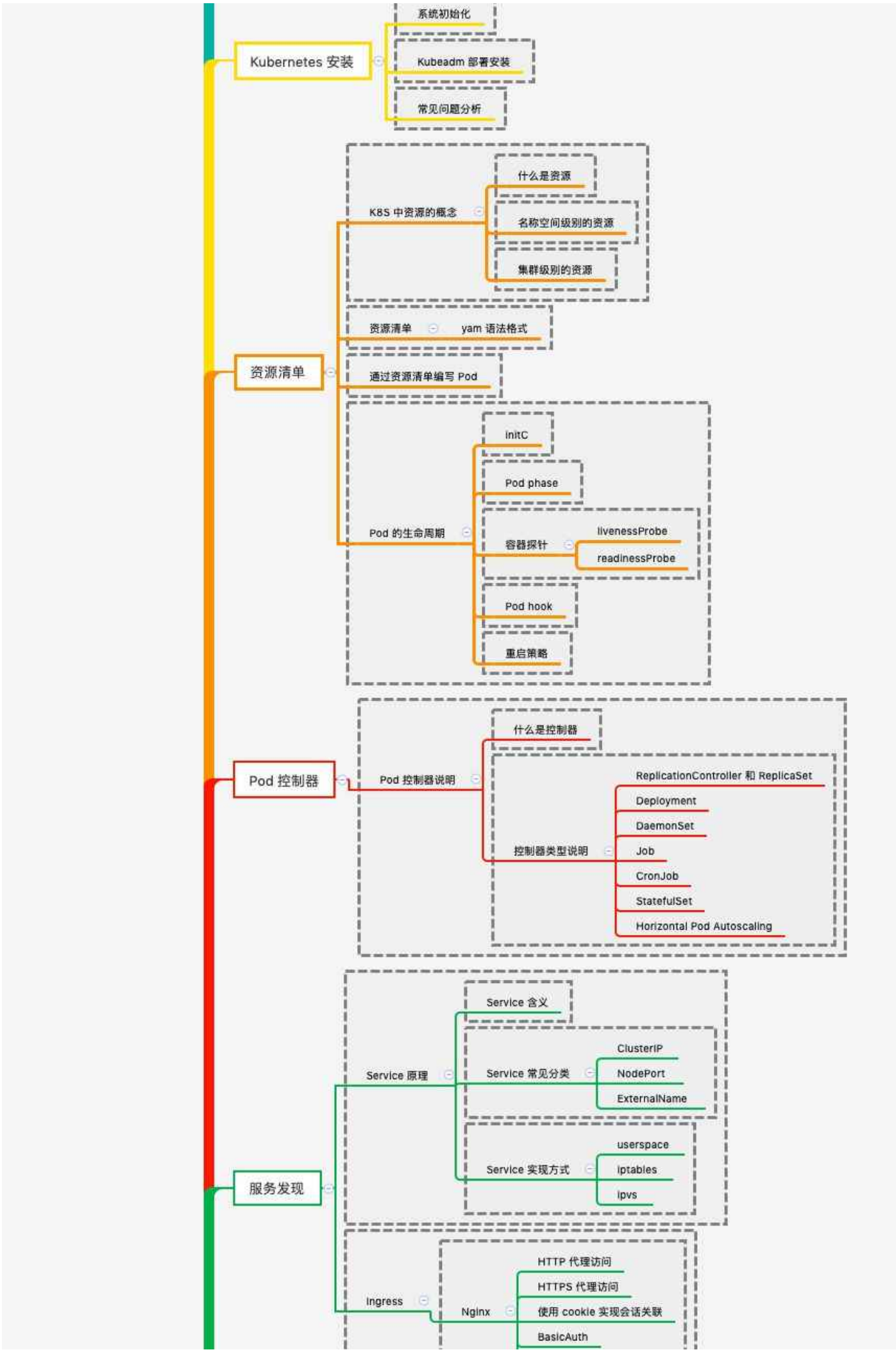
- 密钥与配置管理
  - Kubernetes 允许您存储和管理敏感信息，例如密码、OAuth 令牌和 ssh 密钥。您可以在不重建容器镜像的情况下部署和更新密钥和应用程序配置，也无需在堆栈配置中暴露密钥。

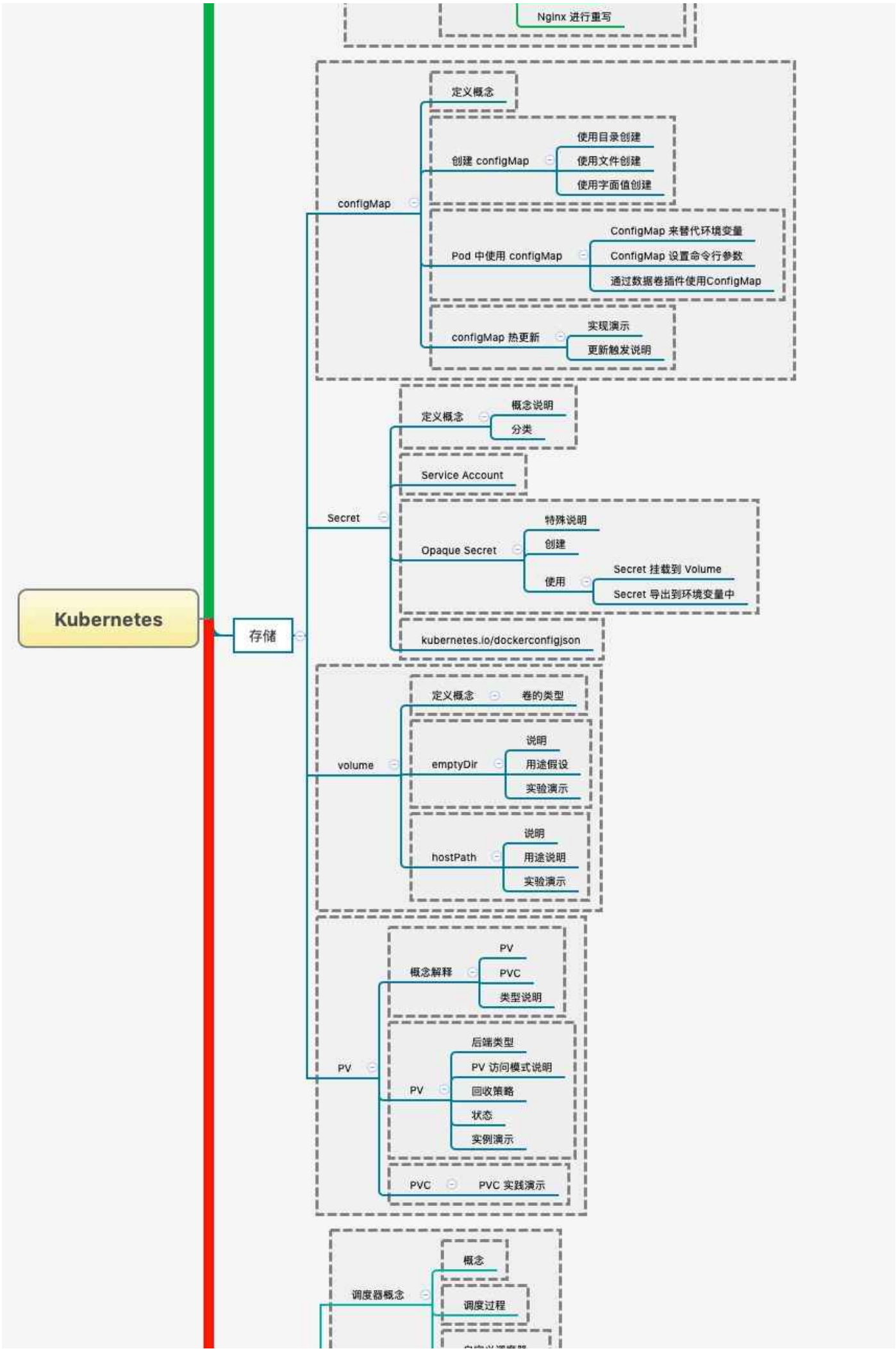
Kubernetes 的出现不仅主宰了容器编排的市场，更改变了过去的运维方式，不仅将开发与运维之间边界变得更加模糊，而且让 DevOps 这一角色变得更加清晰，每一个软件工程师都可以通过 Kubernetes 来定义服务之间的拓扑关系、线上的节点个数、资源使用量并且能够快速实现水平扩容、蓝绿部署等在过去复杂的运维操作。

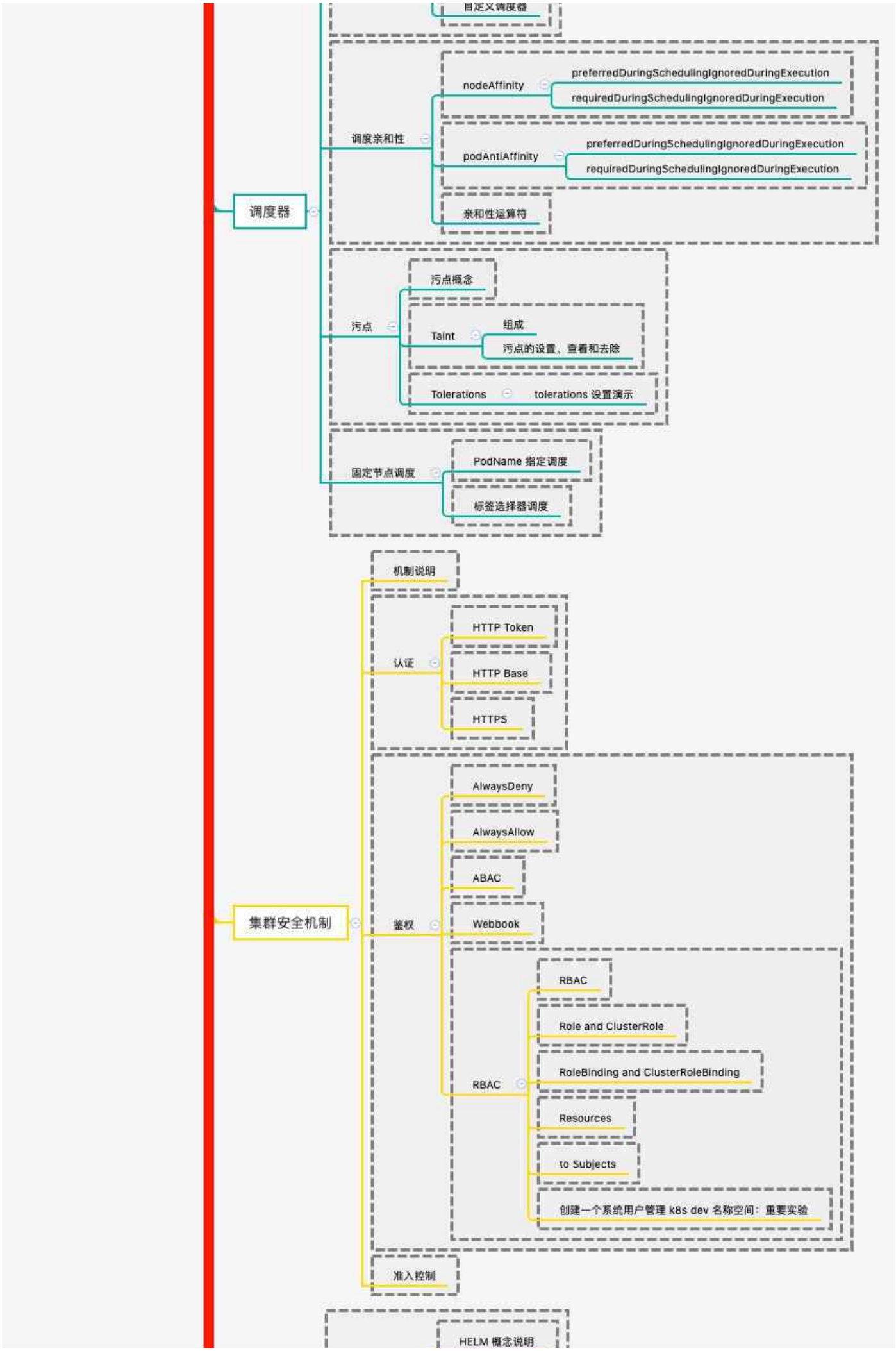
知识图谱

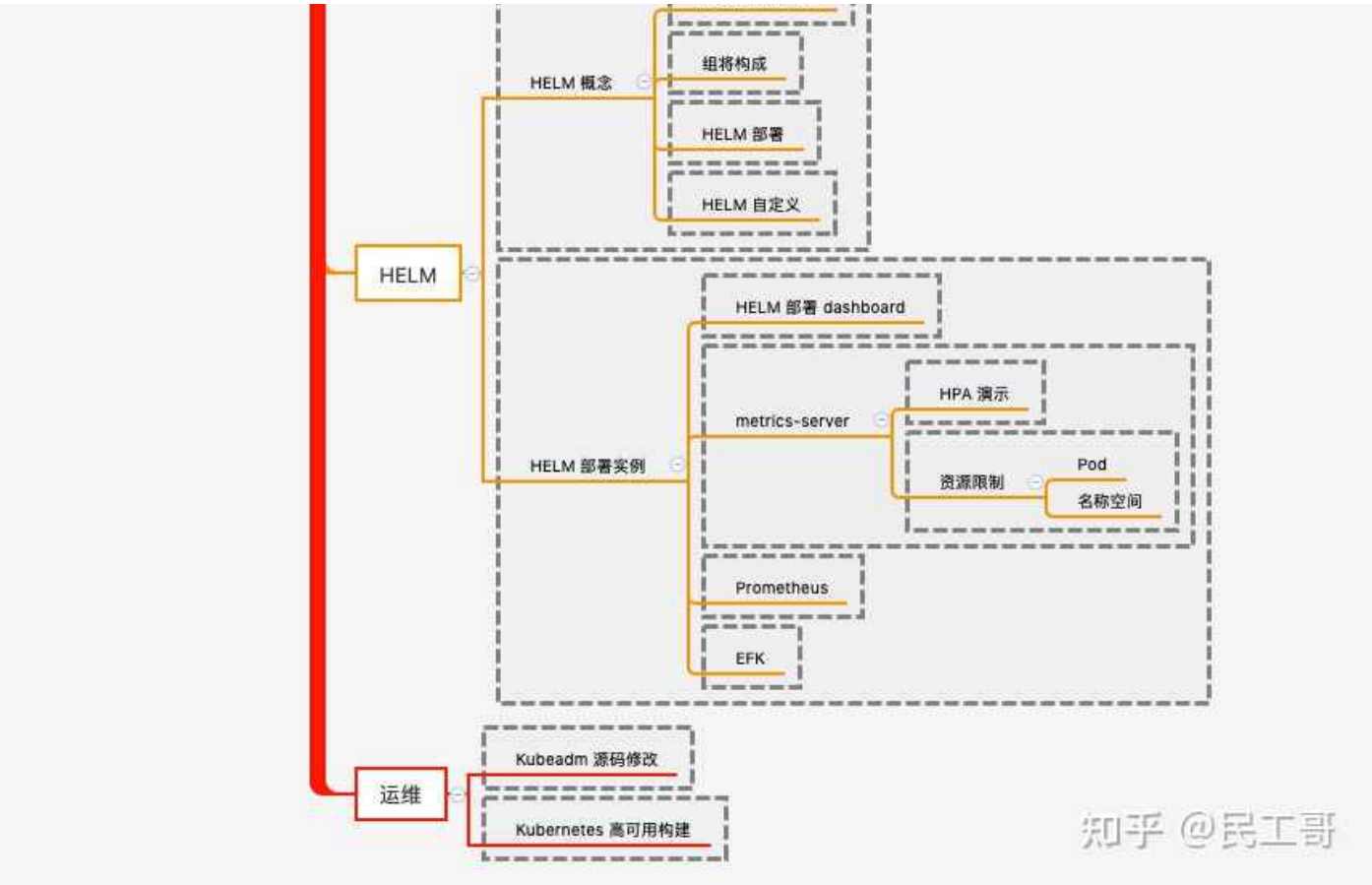
主要介绍学习一些什么知识




Kubernetes 安装
系统初始化
Kubeadm 部署安装
常见问题分析
资源清单
K8S 中资源的概念
什么是资源
名称空间级别的资源
集群级别的资源
资源清单
yam 语法格式
通过资源清单编写 Pod
Pod 的生命周期
initC
Pod phase
容器探针
livenessProbe
readinessProbe
Pod hook
重启策略
Pod 控制器
Pod 控制器说明
什么是控制器
控制器类型说明
ReplicationController 和 ReplicaSet
Deployment
DaemonSet
Job
CronJob
StatefulSet
Horizontal Pod Autoscaling
服务发现
Service 原理
Service 含义
Service 常见分类
ClusterIP
NodePort
ExternalName
Service 实现方式
userspace
iptables
ipvs
Ingress
Nginx
HTTP 代理访问
HTTPS 代理访问
使用 cookie 实现会话关联
BasicAuth

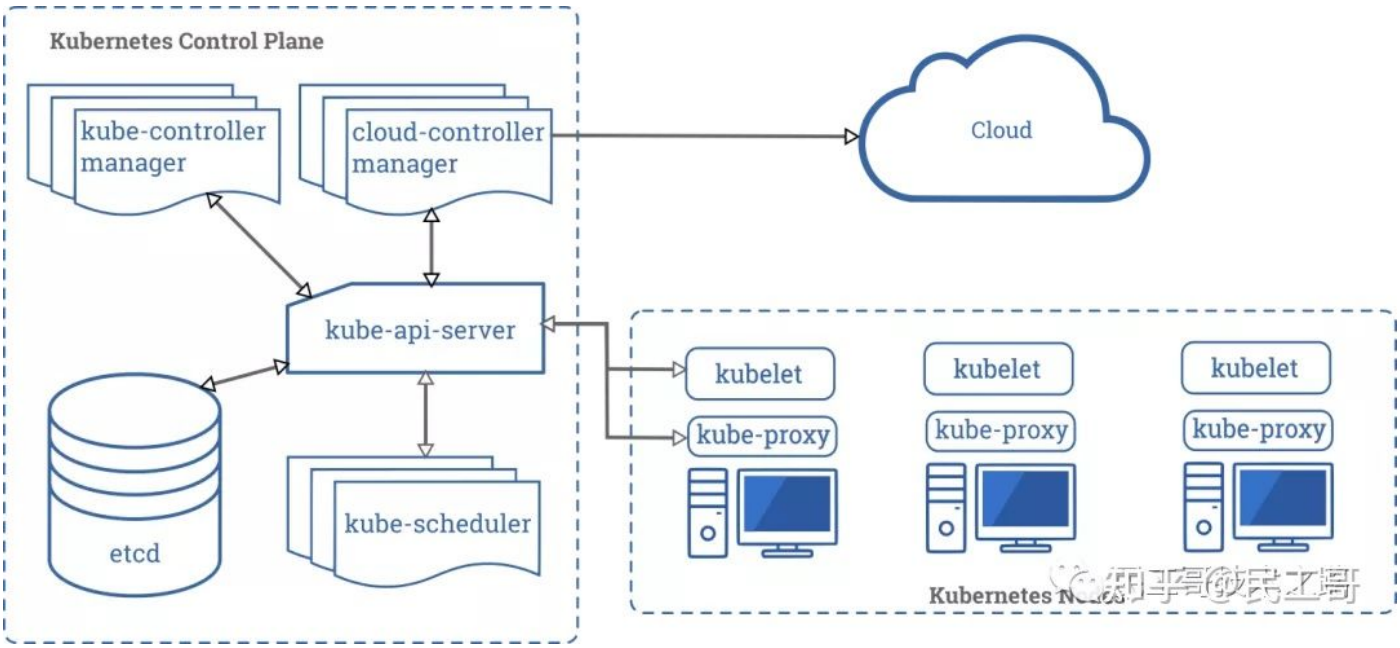

Kubernetes
存储
Nginx 进行重写
configMap
定义概念
创建 configMap
使用目录创建
使用文件创建
使用字面值创建
Pod 中使用 configMap
ConfigMap 来替代环境变量
ConfigMap 设置命令行参数
通过数据卷插件使用ConfigMap
configMap 热更新
实现演示
更新触发说明
Secret
定义概念
概念说明
分类
Service Account
Opaque Secret
特殊说明
创建
使用
Secret 挂载到 Volume
Secret 导出到环境变量中
kubernetes.io/dockerconfigjson
volume
定义概念
卷的类型
emptyDir
说明
用途假设
实验演示
hostPath
说明
用途说明
实验演示
PV
概念解释
PV
PVC
类型说明
PV
后端类型
PV 访问模式说明
回收策略
状态
实例演示
PVC
PVC 实践演示
调度器概念
概念
调度过程


自定义调度器
调度器
调度亲和性
nodeAffinity
preferredDuringSchedulingIgnoredDuringExecution
requiredDuringSchedulingIgnoredDuringExecution
podAntiAffinity
preferredDuringSchedulingIgnoredDuringExecution
requiredDuringSchedulingIgnoredDuringExecution
亲和性运算符
污点
污点概念
Taint
组成
污点的设置、查看和去除
Tolerations
tolerations 设置演示
固定节点调度
PodName 指定调度
标签选择器调度
集群安全机制
机制说明
认证
HTTP Token
HTTP Base
HTTPS
鉴权
AlwaysDeny
AlwaysAllow
ABAC
Webbook
RBAC
RBAC
Role and ClusterRole
RoleBinding and ClusterRoleBinding
Resources
to Subjects
创建一个系统用户管理 k8s dev 名称空间：重要实验
准入控制
HELM 概念说明



软件架构

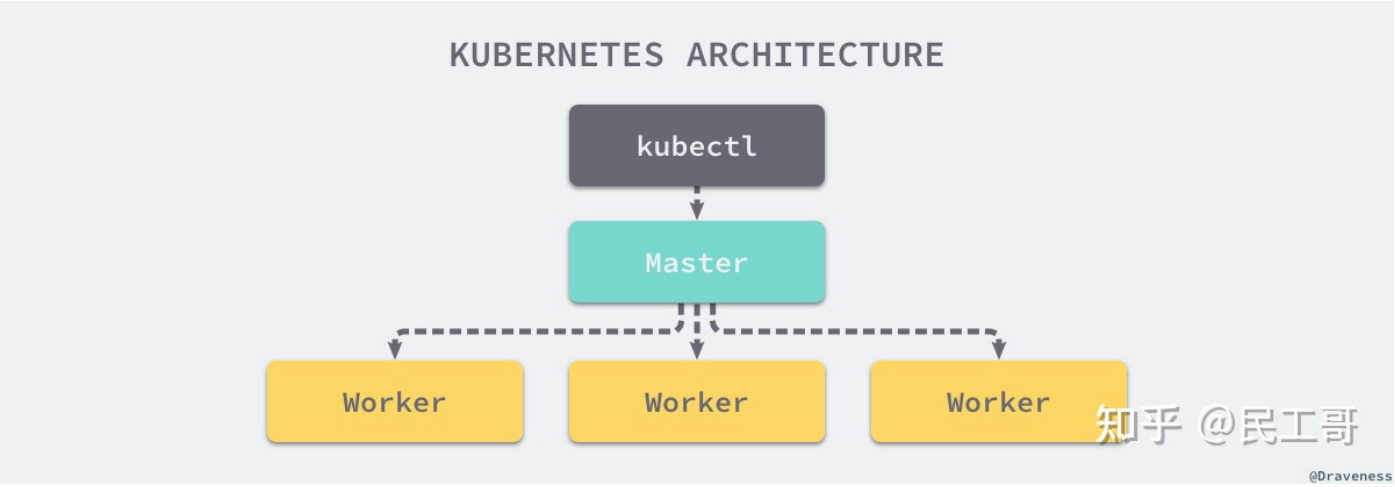
传统的客户端服务端架构

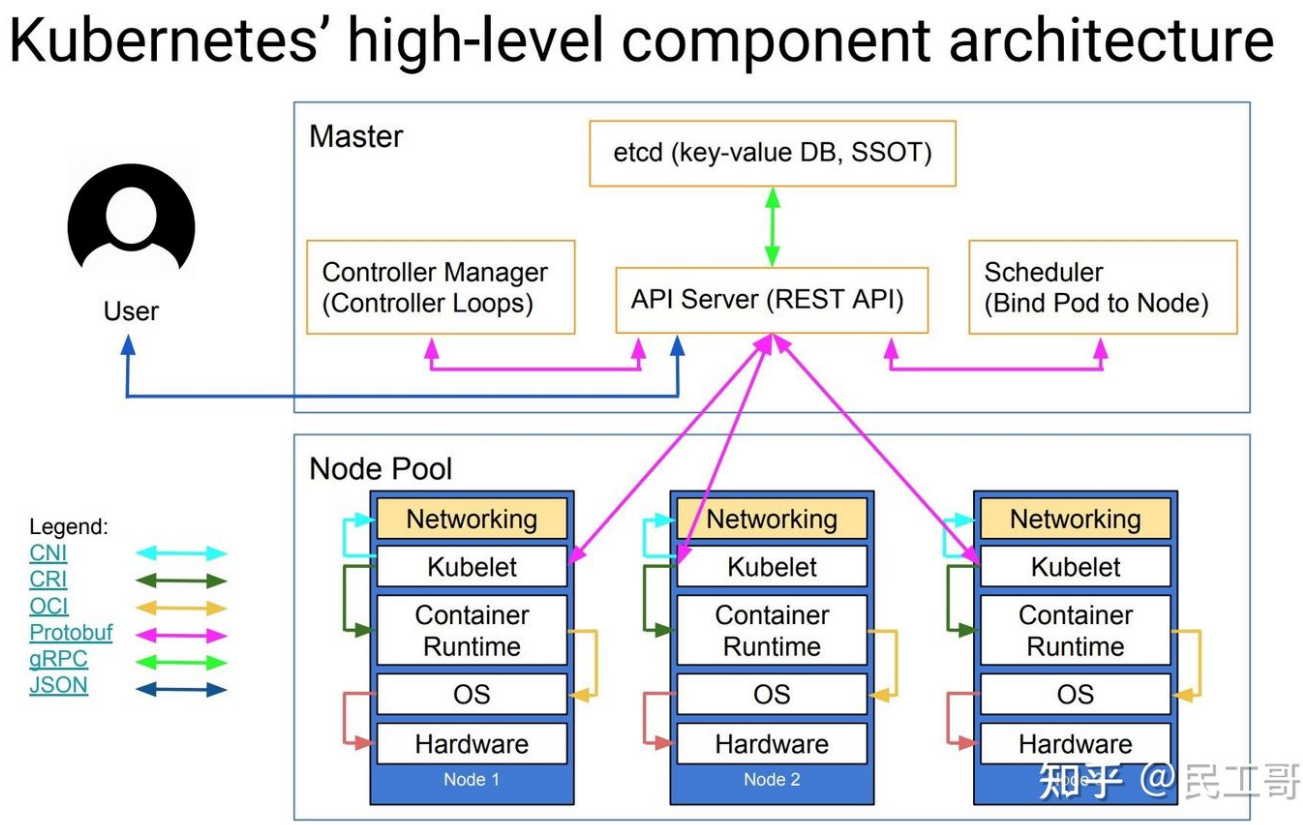


- 架构说明

Kubernetes 遵循非常传统的客户端/服务端的架构模式，客户端可以通过 RESTful 接口或者直接使用 kubectl 与 Kubernetes 集群进行通信，这两者在实际上并没有太多的区别，后者也只是对 Kubernetes 提供的 RESTful API 进行封装并提供出来。每一个 Kubernetes 集群都是由一组 Master

节点和一系列的 Worker 节点组成，其中 Master 节点主要负责存储集群的状态并为 Kubernetes 对象分配和调度资源。





- 主节点服务 - Master 架构

作为管理集群状态的 Master 节点，它主要负责接收客户端的请求，安排容器的执行并且运行控制循环，将集群的状态向目标状态进行迁移。Master 节点内部由下面三个组件构成：

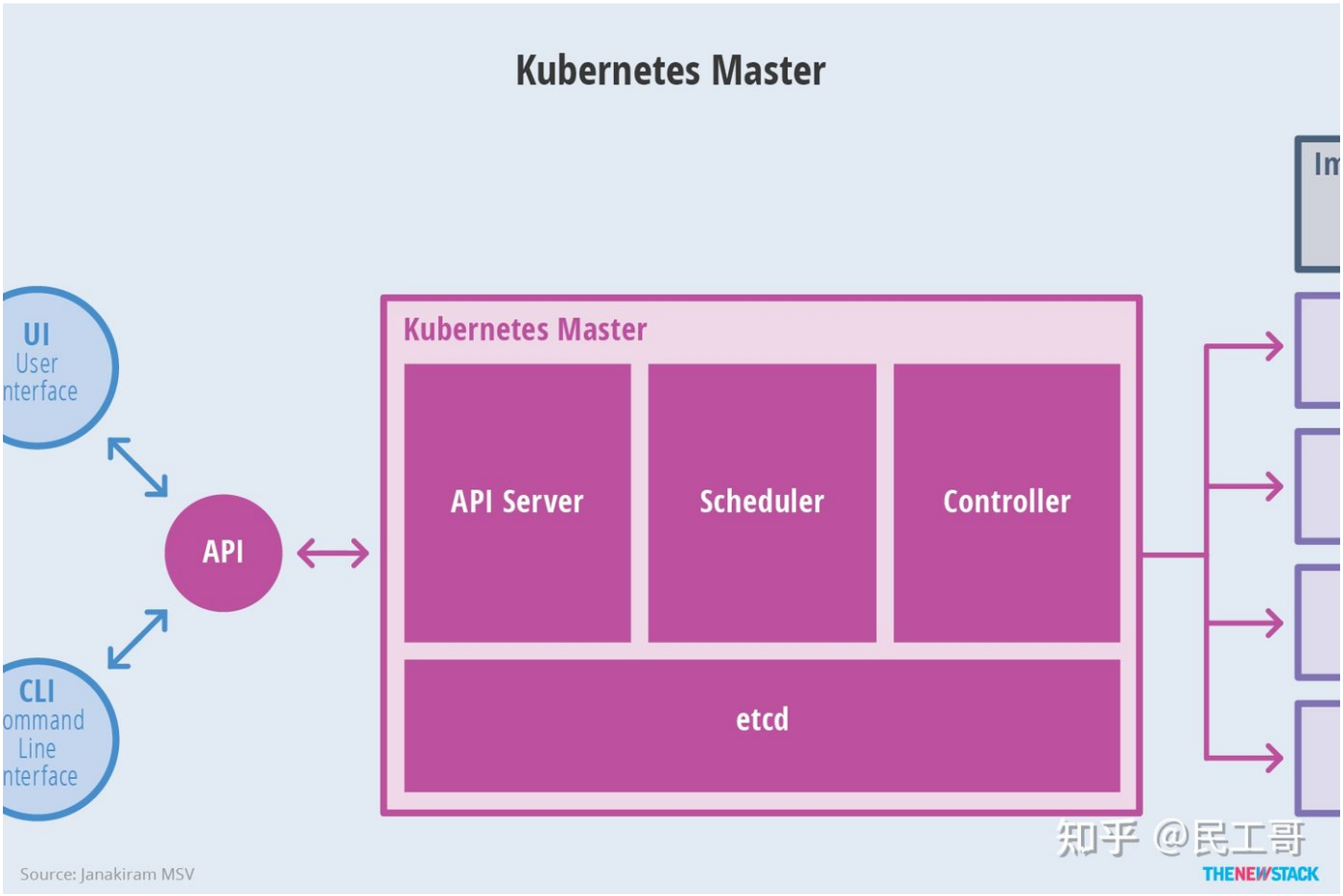
API Server: 负责处理来自用户的请求，其主要作用就是对外提供 RESTful 的接口，包括用于查看集群状态的读请求以及改变集群状态的写请求，也是唯一一个与 etcd 集群通信的组件。

etcd: 是兼具一致性和高可用性的键值数据库，可以作为保存 Kubernetes 所有集群数据的后台数据库。

Scheduler: 主节点上的组件，该组件监视那些新创建的未指定运行节点的 Pod，并选择节点让 Pod 在上面运行。调度决策考虑的因素包括单个 Pod 和 Pod 集合的资源需求、硬件/软件/策略约束、亲

和性和反亲和性规范、数据位置、工作负载间的干扰和最后时限。

controller-manager: 在主节点上运行控制器的组件，从逻辑上讲，每个控制器都是一个单独的进程，但是为了降低复杂性，它们都被编译到同一个可执行文件，并在一个进程中运行。这些控制器包括：节点控制器(负责在节点出现故障时进行通知和响应)、副本控制器(负责为系统中的每个副本控制器对象维护正确数量的 Pod)、端点控制器(填充端点 Endpoints 对象，即加入 Service 与 Pod))、服务帐户和令牌控制器(为新的命名空间创建默认帐户和 API 访问令牌)。



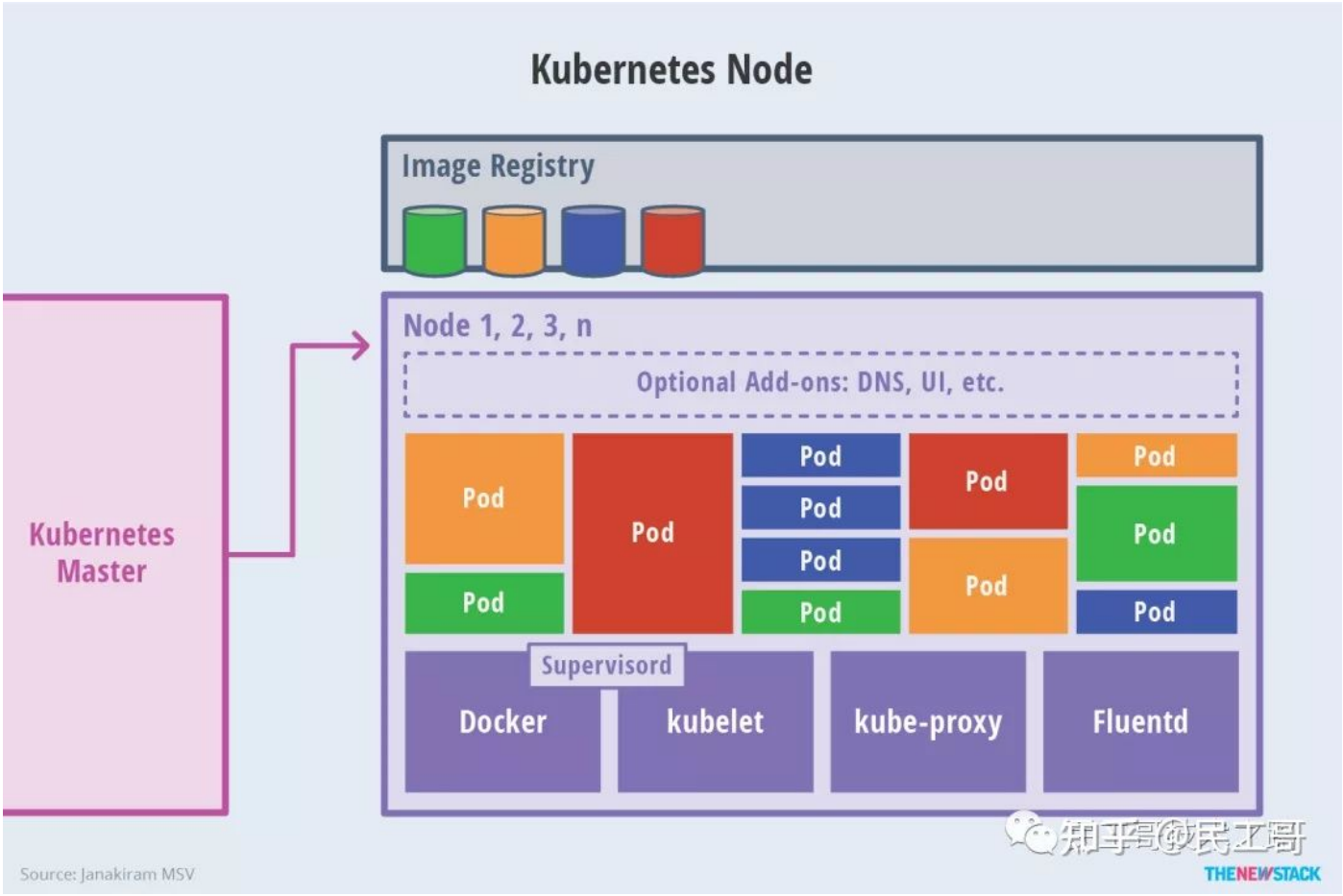
- 工作节点 - Node 架构

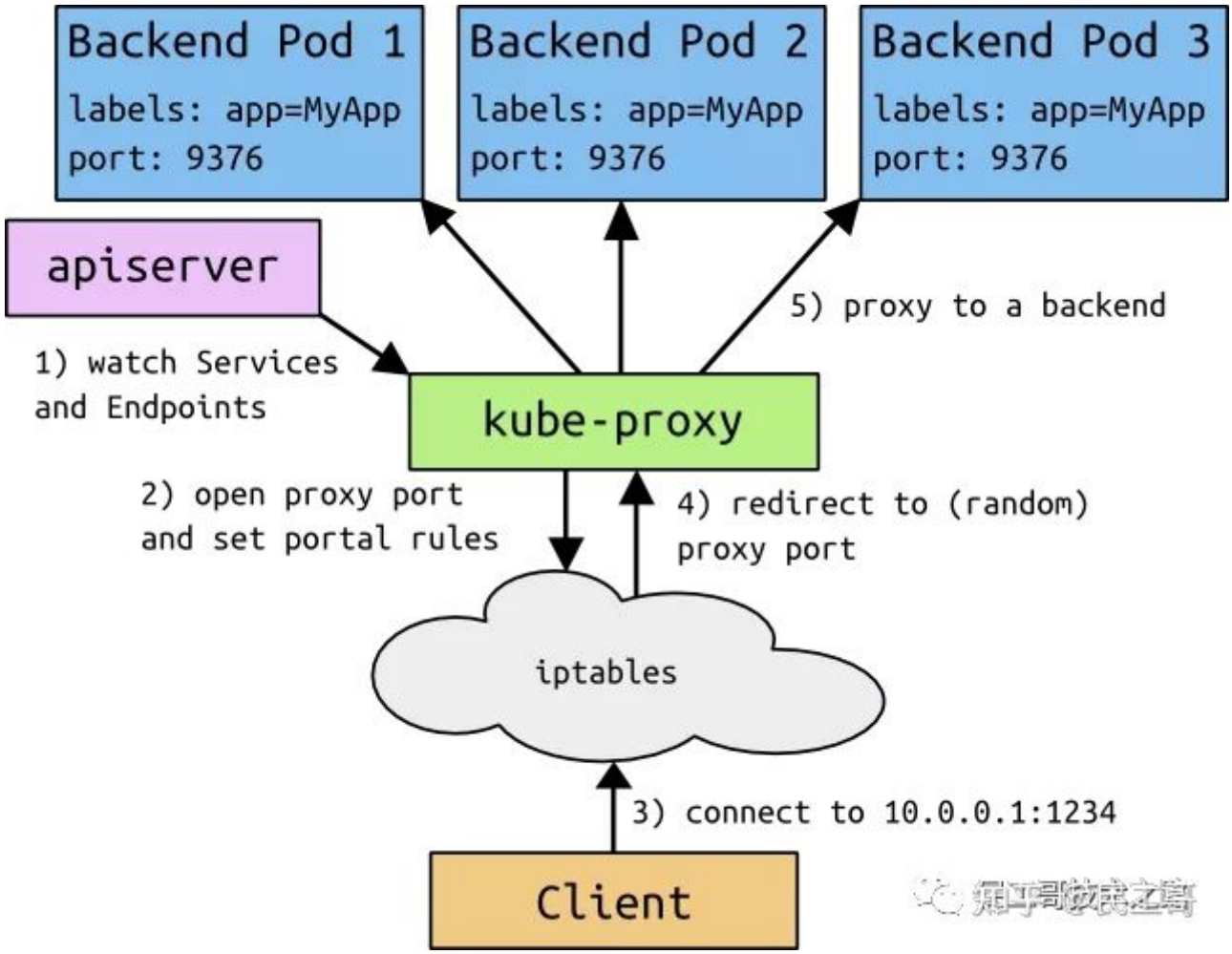
其他的 Worker 节点实现就相对比较简单了，它主要由 kubelet 和 kube-proxy 两部分组成。

kubelet: 是工作节点执行操作的 agent，负责具体的容器生命周期管理，根据从数据库中获取的信息来管理容器，并上报 pod 运行状态等。

kube-proxy: 是一个简单的网络访问代理，同时也是一个 Load Balancer。它负责将访问到某个服务的请求具体分配给工作节点上同一类标签的 Pod。kube-proxy 实质就是通过操作防火墙规则(iptables或者ipvs)来实现 Pod 的映射。

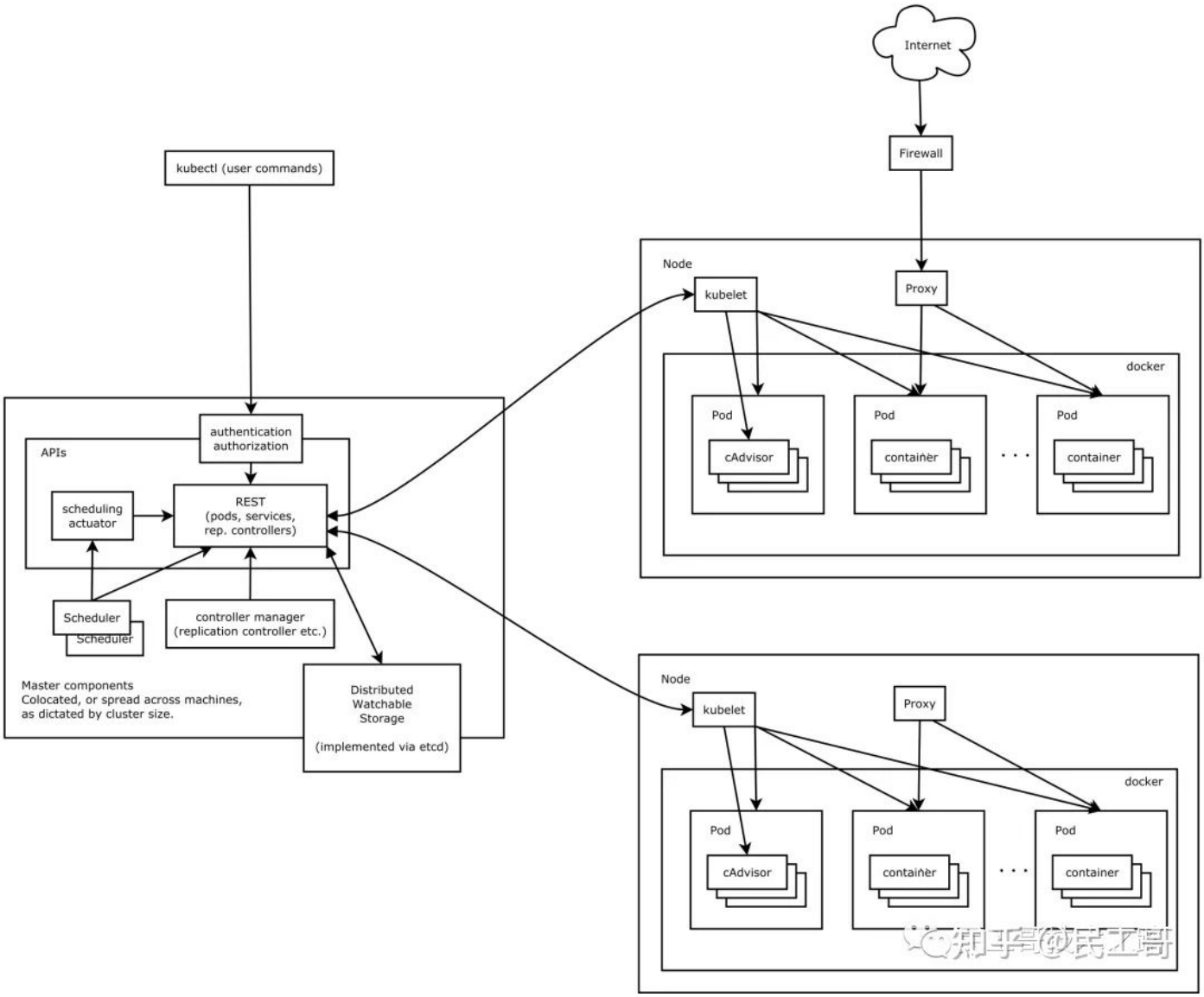
Container Runtime: 容器运行环境是负责运行容器的软件，Kubernetes 支持多个容器运行环境: Docker、 containerd、cri-o、 rktlet 以及任何实现 Kubernetes CRI(容器运行环境接口)。


Kubernetes Node
Image Registry
Node 1, 2, 3, n
Optional Add-ons: DNS, UI, etc.
Kubernetes Master
Pod
Supervisord
Docker
kubelet
kube-proxy
Fluentd
Source: Janakiram MSV
THENEWSTACK


Backend Pod 1
labels: app=MyApp
port: 9376
Backend Pod 2
labels: app=MyApp
port: 9376
Backend Pod 3
labels: app=MyApp
port: 9376
apiserver
1) watch Services
and Endpoints
kube-proxy
5) proxy to a backend
2) open proxy port
and set portal rules
4) redirect to (random)
proxy port
iptables
3) connect to 10.0.0.1:1234
Client

组件说明

主要介绍关于 K8s 的一些基本概念



主要由以下几个核心组件组成：

- apiserver
  - 所有服务访问的唯一入口，提供认证、授权、访问控制、API 注册和发现等机制

- controller manager
  - 负责维护集群的状态，比如副本期望数量、故障检测、自动扩展、滚动更新等
- scheduler
  - 负责资源的调度，按照预定的调度策略将 Pod 调度到相应的机器上

- etcd
  - 键值对数据库，保存了整个集群的状态
- kubelet
  - 负责维护容器的生命周期，同时也负责 Volume 和网络的管理

- kube-proxy
  - 负责为 Service 提供 cluster 内部的服务发现和负载均衡
- Container runtime
  - 负责镜像管理以及 Pod 和容器的真正运行

除了核心组件，还有一些推荐的插件：

- CoreDNS
  - 可以为集群中的 SVC 创建一个域名 IP 的对应关系解析的 DNS 服务

- Dashboard
  - 给 K8s 集群提供了一个 B/S 架构的访问入口
- Ingress Controller
  - 官方只能够实现四层的网络代理，而 Ingress 可以实现七层的代理

- Prometheus
  - 给 K8s 集群提供资源监控的能力
- Federation
  - 提供一个可以跨集群中心多 K8s 的统一管理功能，提供跨可用区的集群

以上内容参考链接: https://www.escapelife.site/posts/2c4214e7.html

安装

安装v1.16.0版本，竟然成功了。记录在此，避免后来者踩坑。

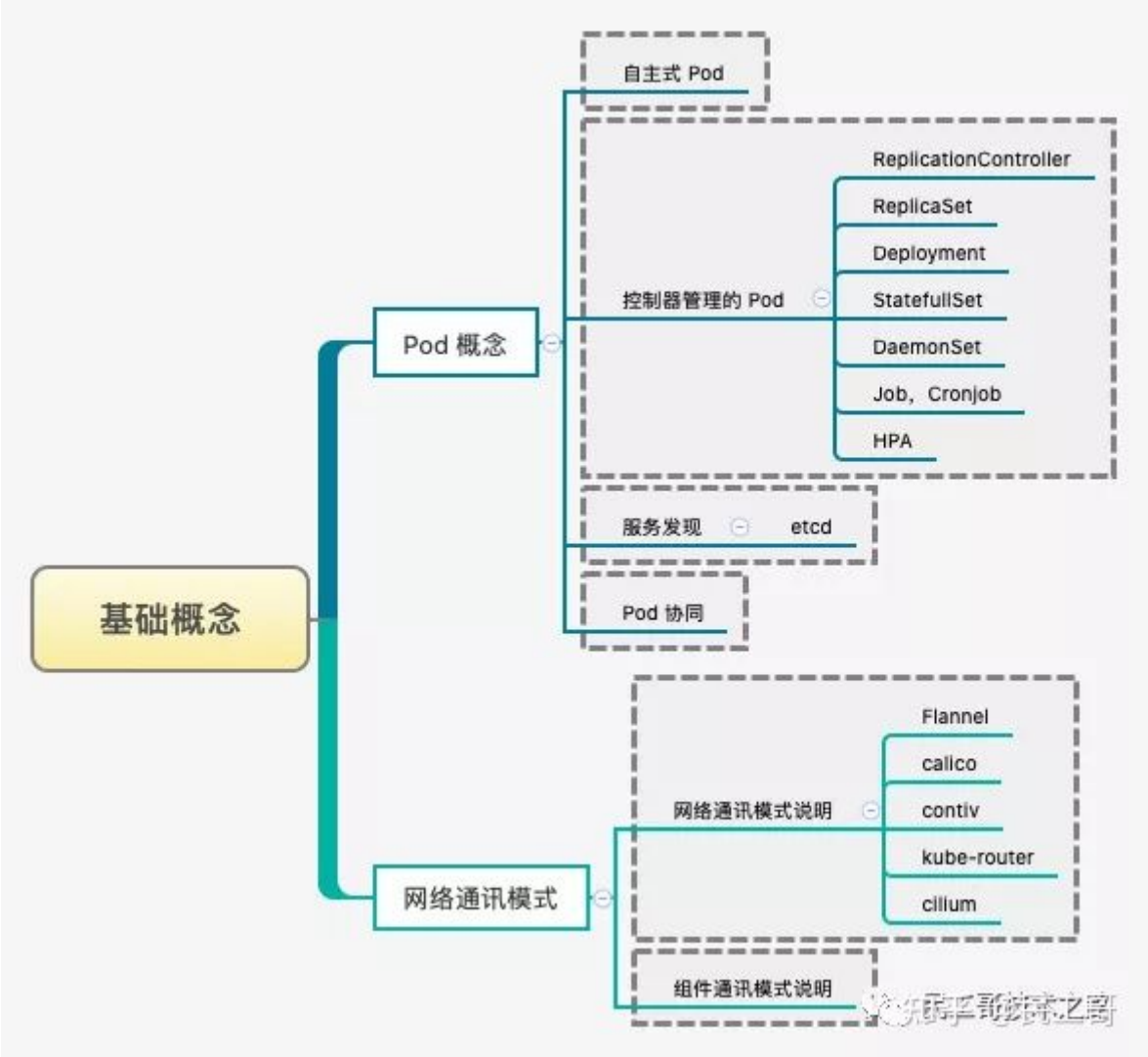
本篇文章，安装大步骤如下：

- 安装docker-ce 18.09.9（所有机器）
- 设置k8s环境前置条件（所有机器）
- 安装k8s v1.16.0 master管理节点
- 安装k8s v1.16.0 node工作节点
- 安装flannel（master）

详细安装步骤参考：CentOS 搭建 K8S，一次性成功，收藏了！ 集群安装教程请参考：全网最新、最详细基于V1.20版本，无坑部署最小化 K8S 集群教程

Pod 实现原理

Pod 就是最小并且最简单的 Kubernetes 对象



Pod、Service、Volume 和 Namespace 是 Kubernetes 集群中四大基本对象，它们能够表示系统中部署的应用、工作负载、网络和磁盘资源，共同定义了集群的状态。Kubernetes 中很多其他的资源其实只对这些基本的对象进行了组合。

- Pod -> 集群中的基本单元
- Service -> 解决如何访问 Pod 里面服务的问题
- Volume -> 集群中的存储卷
- Namespace -> 命名空间为集群提供虚拟的隔离作用

详细介绍请参考：Kubernetes 之 Pod 实现原理

Harbor 仓库

Kuternetes 企业级 Docker 私有仓库 Harbor 工具。 Harbor 的每个组件都是以 Docker 容器的形式构建的，使用 Docker Compose 来对它进行部署。用于部署 Harbor 的 Docker Compose 模板位于 /Deployer/docker-compose.yml 中，其由 5 个容器组成，这几个容器通过 Docker link 的形式连接在一起，在容器之间通过容器名字互相访问。对终端用户而言，只需要暴露 Proxy(即Nginx) 的服务端口即可。

- Proxy

    - 由Nginx服务器构成的反向代理

- Registry
    - 由Docker官方的开源官方的开源Registry镜像构成的容器实例
- UI
    - 即架构中的core services服务，构成此容器的代码是Harbor项目的主体

- MySQL
    - 由官方MySQL镜像构成的数据库容器
- Log
    - 运行着rsyslogd的容器，通过log-driver的形式收集其他容器的日志

详细介绍与搭建步骤请参考：企业级环境中基于 Harbor 搭建

YAML 语法

YAML 是一种非常简洁/强大/专门用来写配置文件的语言！

YAML 全称是 ”YAML Ain’ t a Markup Language” 的递归缩写，该语言的设计参考了 JSON / XML 和 SDL 等语言,强调以数据为中心，简洁易读，编写简单。

YAML 语法特性

学过编程的人理解起来应该非常容易

XML	JSON	YAML

<Servers>
<Server>
<name>Server1</name>
<owner>John</owner>
<created>123456</created>
<status>active</status>
</Server>
</Servers>
	{
	Servers: [
	{
	name: Server1,
	owner: John,
	created: 123456,
	status: active
	}
	]
	}
		Servers:
		- name: Server1
		owner: John
		created: 123456
		status: active

语法特点

- 大小写敏感
- 通过缩进表示层级关系
- 禁止使用tab缩进，只能使用空格键
- 缩进的空格数目不重要，只要相同层级左对齐
- 使用#表示注释

推荐给大家一篇文章：Kubernetes 之 YAML 语法，这篇文章介绍的非常详细，有很多例子说明。

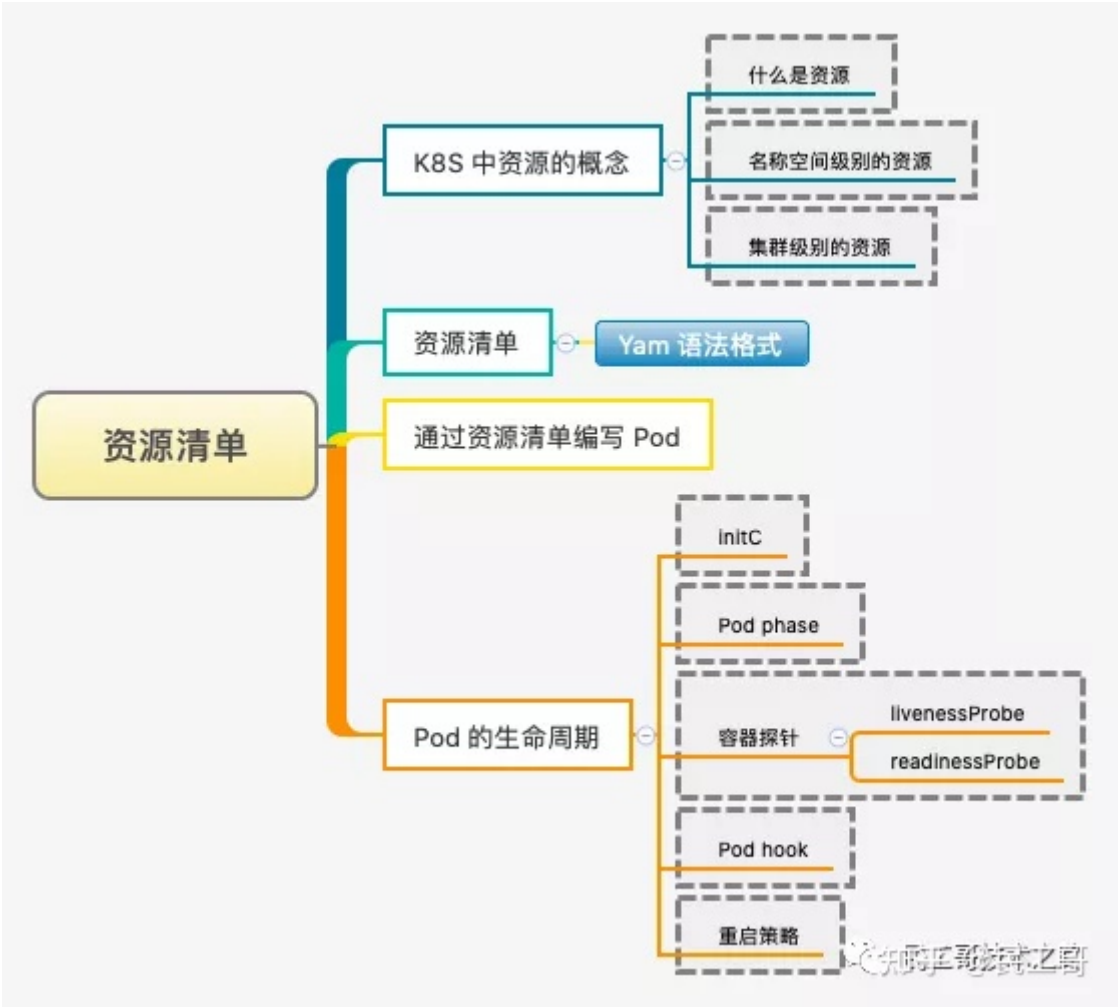
资源清单

K8S 中所有的内容都抽象为了资源，资源实例化之后就叫做对象。

在 Kubernetes 系统中，Kubernetes 对象是持久化的实体，Kubernetes 使用这些实体去表示整个集群的状态。特别地，它们描述了如下信息：

- 哪些容器化应用在运行，以及在哪个 Node 上
- 可以被应用使用的资源
- 关于应用运行时表现的策略，比如重启策略、升级策略，以及容错策略

Kubernetes 对象是 “目标性记录” —— 一旦创建对象，Kubernetes 系统将持续工作以确保对象存在。通过创建对象，本质上是在告知 Kubernetes 系统，所需要的集群工作负载看起来是什么样子的，这就是 Kubernetes 集群的期望状态。

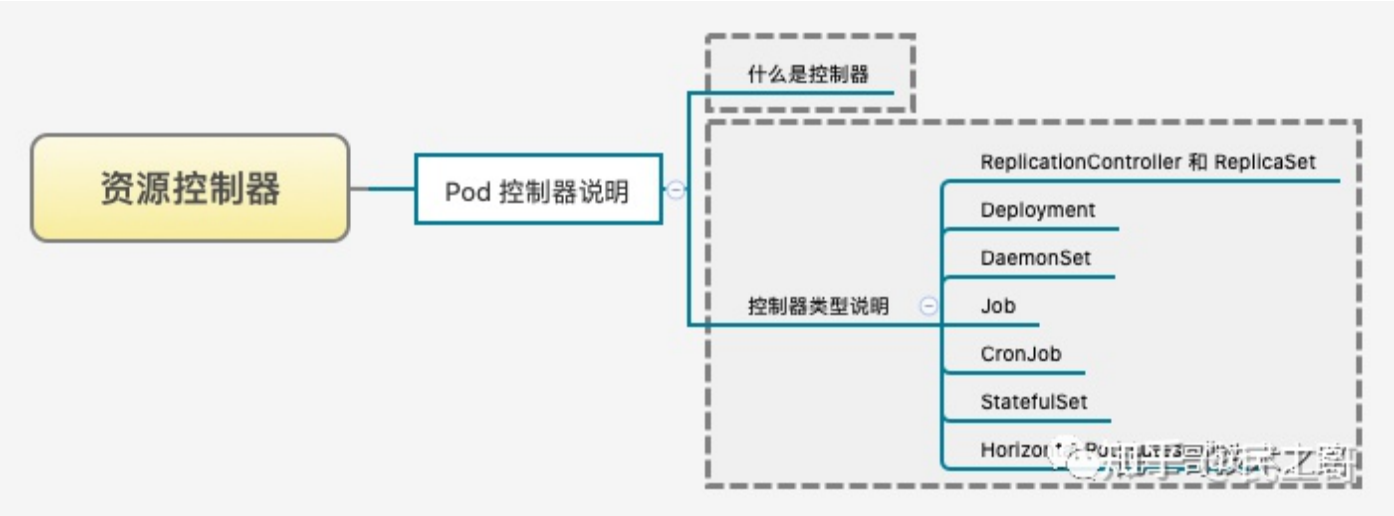


Kubernetes 之资源清单详细介绍看这里

资源控制器

Kubernetes 资源控制器配置文件的编写是学习 K8S 的重中之重！

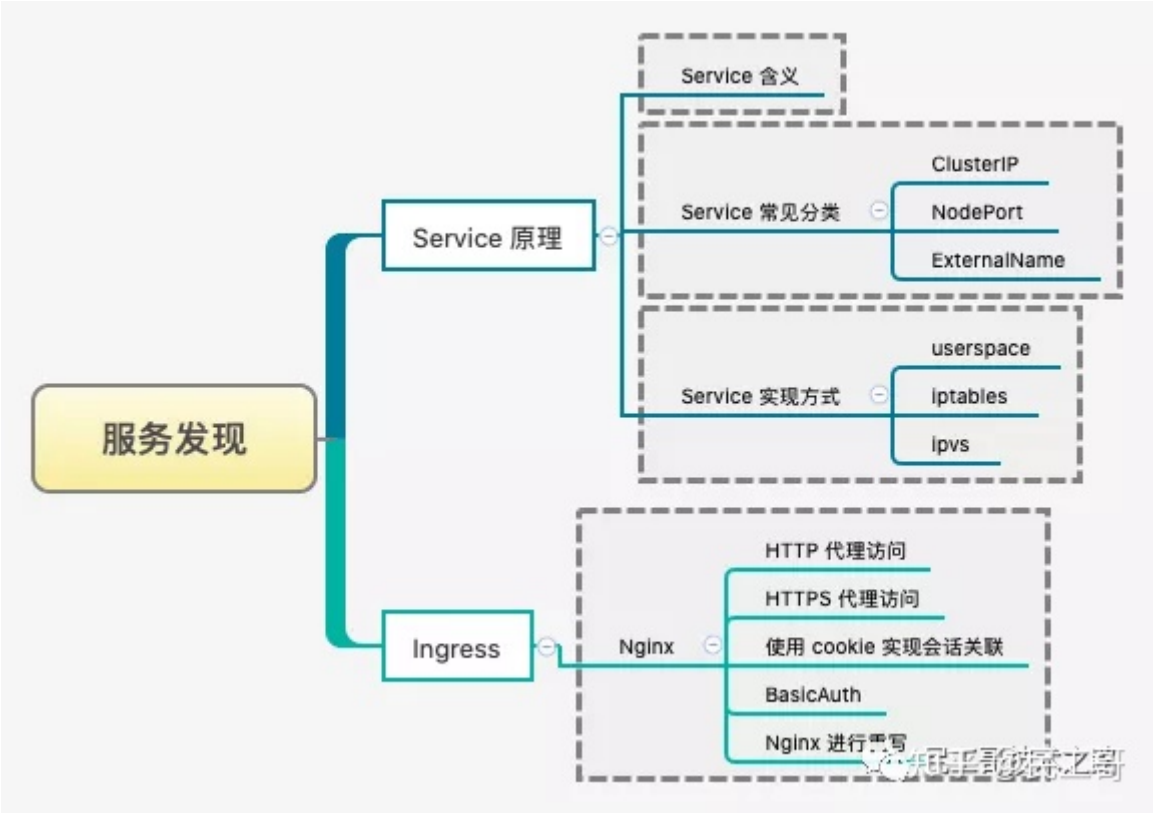
资源配额控制器确保了指定的资源对象始终不会超过配置的资源，能够有效的降低整个系统宕机的机率，增强系统的鲁棒性，对整个集群的稳定性有非常重要的作用。



Kubernetes 资源控制器使用指南手册

服务发现

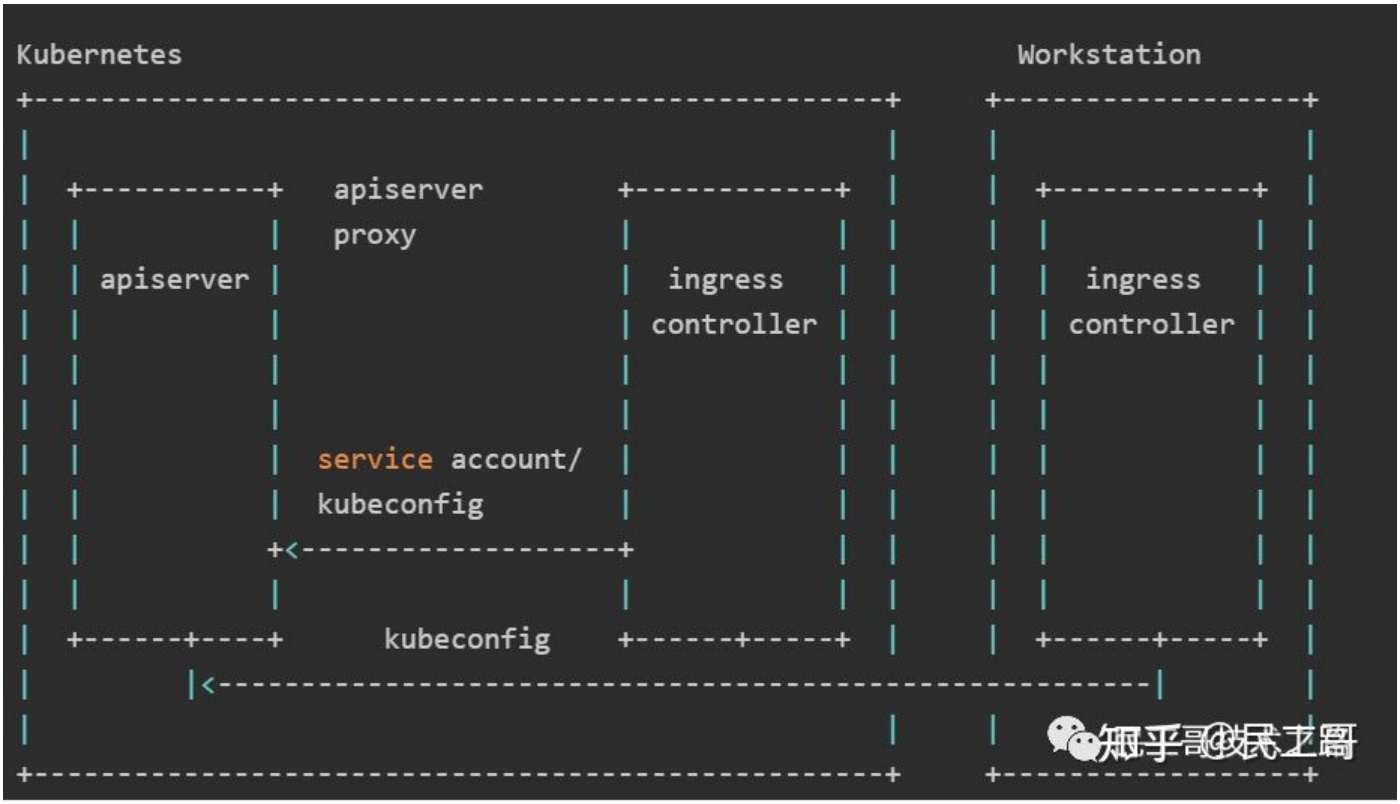
Kubernetes 中为了实现服务实例间的负载均衡和不同服务间的服务发现，创造了 Service 对象，同时又为从集群外部访问集群创建了 Ingress 对象。



Kubernetes 之服务发现

Ingress 服务

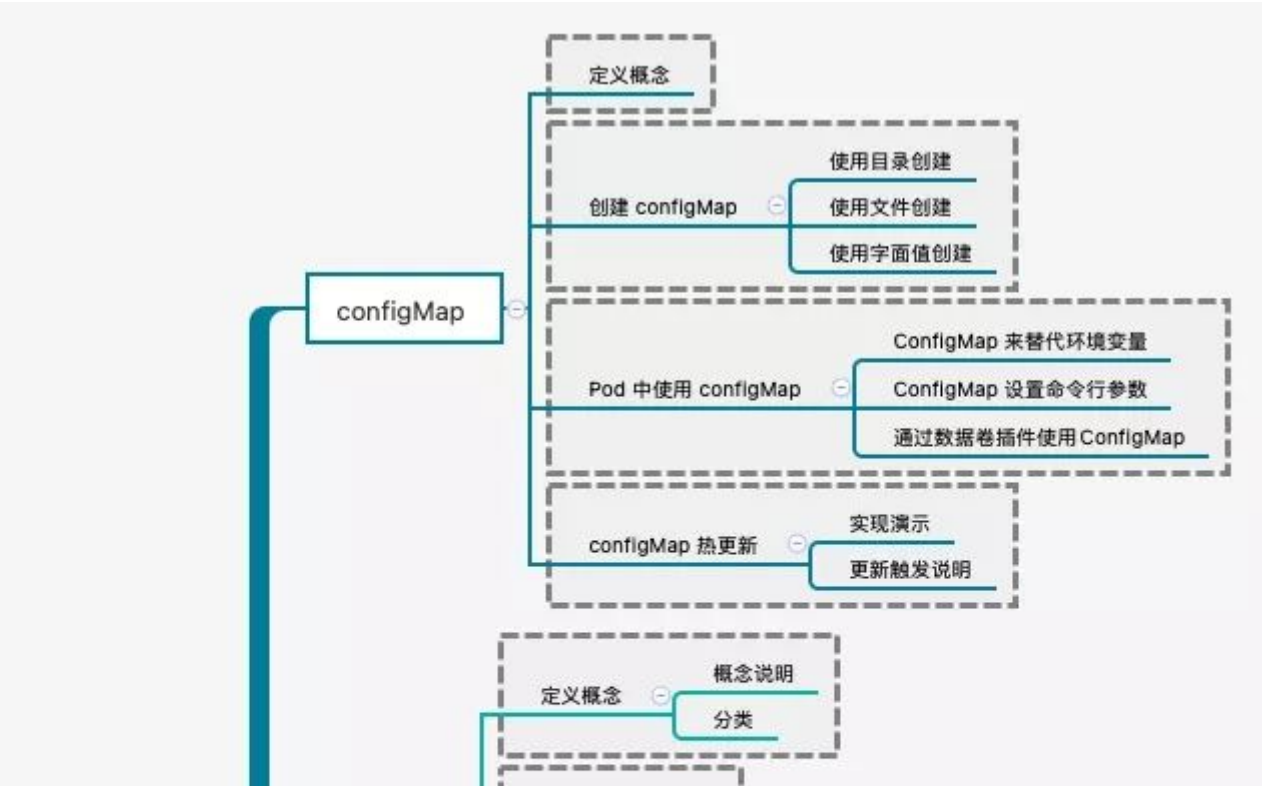
我们都知道传统的 SVC 只支持四层上面的代码，而对于七层上的代码而无能为力。比如：我们使用 K8S 集群对外提供 HTTPS 的服务，为了方便和便捷，我们需要在对外的 Nginx 服务上面配置 SSL 加密，但是将请求发送给后端服务的时候，进行证书卸载的操作，后续都是用 HTTP 的协议进行处理。而面对此问题，K8S 中给出了使用 Ingress (K8S在1.11版本中推出了)来进行处理。

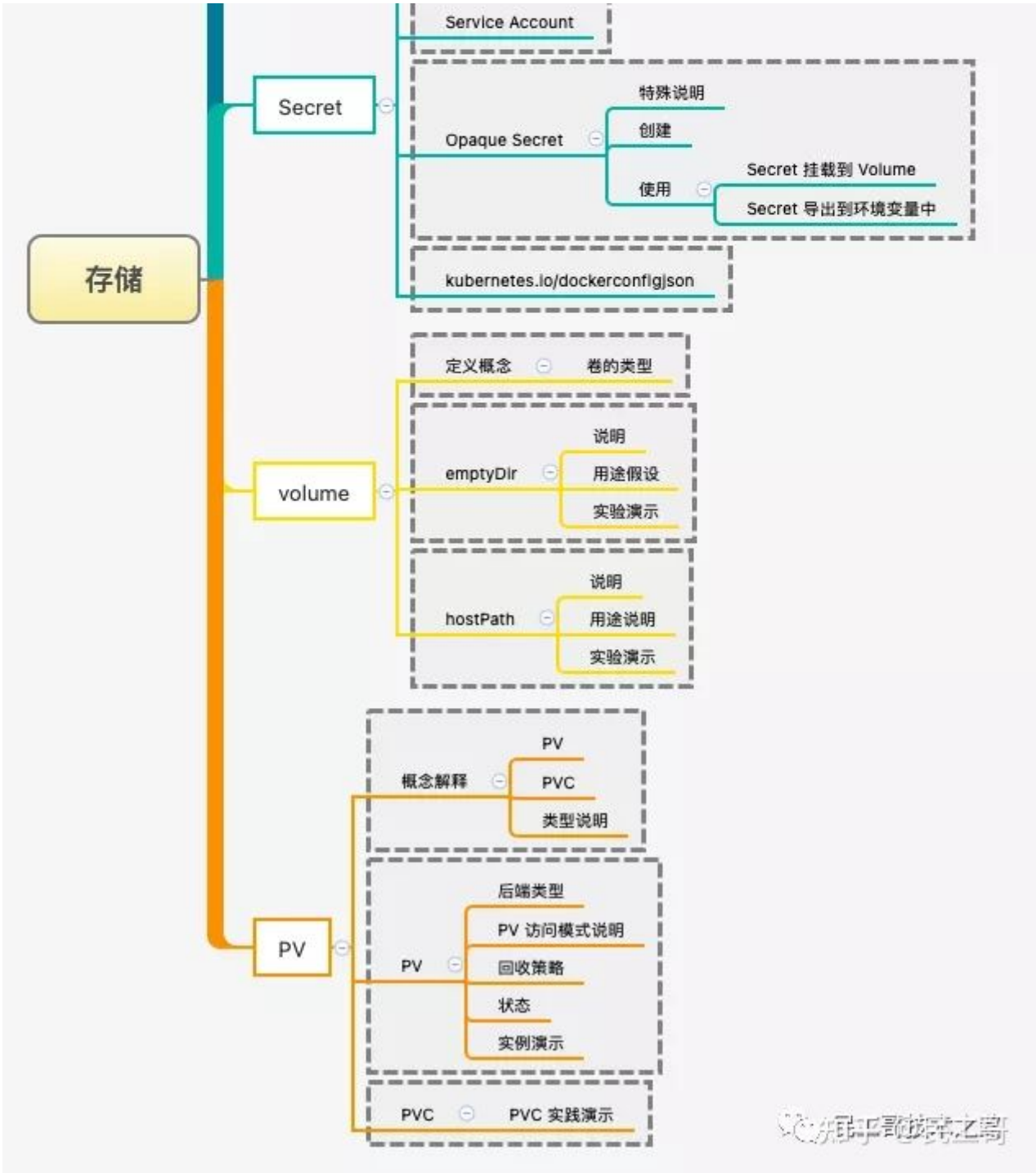


更多详细内容请参阅：Kubernetes 之 Ingress 服务，介绍关于 Ingress 服务的安装方式，配置关于 Ingress 服务的 HTTP 代理访问，介绍 Ingress 服务的 BasicAuth 认证方式，介绍 Ingress 的进行规则重写的方式。

数据存储

在之前的文章中，我们已经知道了很多 K8S 中的组件了，包括资源控制器等。在资源控制器中，我们说到了 StatefulSet 这个控制器组件，其专门为了有状态服务而生的，而对应的存储要存放到哪里呢？





介绍 K8S 中常见的存储机制可以让我们所使用的：Kubernetes 之数据存储

集群调度

有这样一个需求，就是集群中多台服务的配置是不一致的。这就导致资源分配并不是均匀的，比如我们需要有些服务节点用来运行计算密集型的服务，而有些服务节点来运行需要大量内存的服务。而在 k8s 中当然也配置了相关服务来处理上述的问题，那就是 Scheduler。

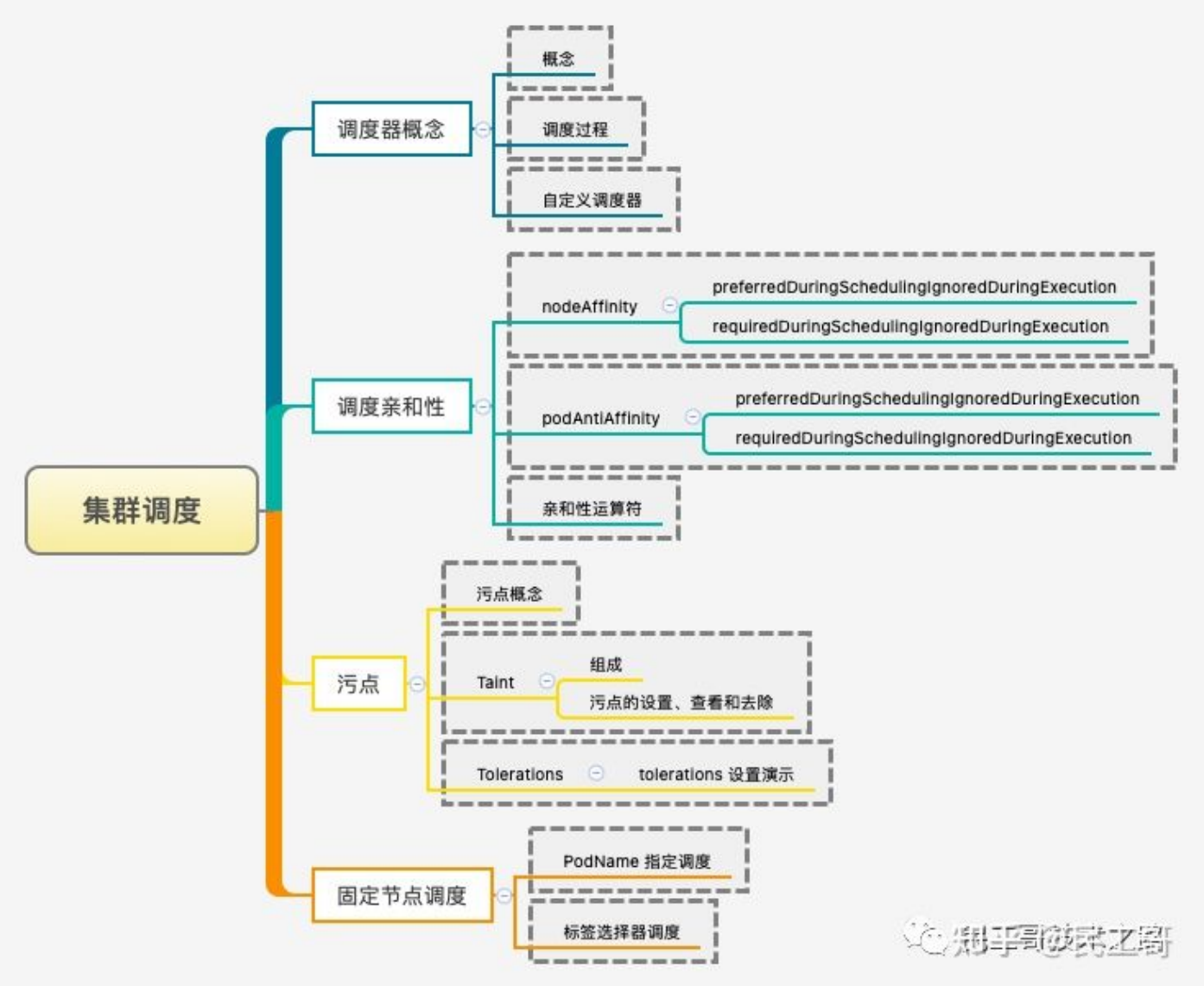
Scheduler 是 kubernetes 的调度器，主要的任务是把定义的 Pod 分配到集群的节点上。听起来非常简单，但有很多要考虑的问题：

- 公平
  - 如何保证每个节点都能被分配资源

- 资源高效利用
  - 集群所有资源最大化被使用
- 效率
  - 调度的性能要好，能够尽快地对大批量的 Pod 完成调度工作

- 灵活
  - 允许用户根据自己的需求控制调度的逻辑

Scheduler 是作为单独的程序运行的，启动之后会一直坚挺 API Server，获取 PodSpec.NodeName 为空的 Pod，对每个 Pod 都会创建一个 binding，表明该 Pod 应该放到哪个节点上。



详细的介绍请参考：Kubernetes 之集群调度

kubectl 使用指南

kubectl 是 Kubernetes 自带的客户端，可以用它来直接操作 Kubernetes 集群。

日常在使用 Kubernetes 的过程中，kubectl 工具可能是最常用的工具了，所以当我们花费大量的时间去研究和学习 Kuernetes 的时候，那么我们就非常有必要去了解下如何高效的使用它了。

从用户角度来说，kubectl 就是控制 Kubernetes 的驾驶舱，它允许你执行所有可能的 Kubernetes 操作；从技术角度来看，kubectl 就是 Kubernetes API 的一个客户端而已。

Kubernetes API 是一个 HTTP REST API 服务，该 API 服务才是 Kubernetes 的真正用到的用户接口，所以 Kubernetes 通过该 API 进行实际的控制。这也就意味着每个 Kubernetes 的操作都会通过 API 端点暴露出去，当然也就可以通过对这些 API 端口进行 HTTP 请求来执行相应的操作。所以，kubectl 最主要的工作就是执行 Kubernetes API 的 HTTP 请求。

工具使用参数

Plain Text

```
get        #显示一个或多个资源
describe   #显示资源详情
create     #从文件或标准输入创建资源
update    #从文件或标准输入更新资源
delete    #通过文件名、标准输入、资源名或者 label 删除资源
log        #输出 pod 中一个容器的日志
rolling-update  #对指定的 RC 执行滚动升级
exec  #在容器内部执行命令
port-forward #将本地端口转发到 Pod
proxy    #为 Kubernetes API server 启动代理服务器
run      #在集群中使用指定镜像启动容器
expose    #将 SVC 或 pod 暴露为新的 kubernetes service
label      #更新资源的 label
config    #修改 kubernetes 配置文件
cluster-info #显示集群信息
api-versions #以”组/版本”的格式输出服务端支持的 API 版本
version        #输出服务端和客户端的版本信息
help          #显示各个命令的帮助信息
ingress-nginx  #管理 ingress 服务的插件(官方安装和使用方式)
```

使用相关配置

Plain Text

```
# Kubectl自动补全
$ source <(kubectl completion zsh)
$ source <(kubectl completion bash)

# 显示合并后的 kubeconfig 配置
$ kubectl config view

# 获取pod和svc的文档
$ kubectl explain pods,svc
```

创建资源对象

分步骤创建

Plain Text

```
# yaml
kubectl create -f xxx-rc.yaml
kubectl create -f xxx-service.yaml

# json
kubectl create -f ./pod.json
cat pod.json | kubectl create -f -

# yaml2json
kubectl create -f docker-registry.yaml --edit -o json
```

一次性创建

Plain Text

```
kubectl create -f xxx-service.yaml -f xxx-rc.yaml
```

根据目录下所有的 yaml 文件定义内容进行创建

Plain Text

```
kubectl create -f <目录>
```

使用 url 来创建资源

Plain Text

```
kubectl create -f https://git.io/vPieo
```

查看资源对象

查看所有 Node 或 Namespace 对象

Plain Text

```
kubectl get nodes
kubectl get namespace
```

查看所有 Pod 对象

Plain Text

```
# 查看子命令帮助信息
kubectl get --help

# 列出默认namespace中的所有pod
kubectl get pods

# 列出指定namespace中的所有pod
kubectl get pods --namespace=test

# 列出所有namespace中的所有pod
kubectl get pods --all-namespaces

# 列出所有pod并显示详细信息
kubectl get pods -o wide
kubectl get replicationcontroller web
kubectl get -k dir/
kubectl get -f pod.yaml -o json
kubectl get rc/web service/frontend pods/web-pod-13je7
kubectl get pods/app-prod-78998bf7c6-ttp9g --namespace=test -o wide
kubectl get -o template pod/web-pod-13je7 --template={{.status.phase}}

# 列出该namespace中的所有pod包括未初始化的
kubectl get pods,rc,services --include-uninitialized
```

查看所有 RC 对象

Plain Text

```
kubectl get rc
```

查看所有 Deployment 对象

Plain Text

```
# 查看全部deployment
kubectl get deployment

# 列出指定deployment
kubectl get deployment my-app
```

查看所有 Service 对象

```
kubectl get svc
kubectl get service
```

查看不同 Namespace 下的 Pod 对象

```
kubectl get pods -n default
kubectl get pods --all-namespace
```

查看资源描述

显示 Pod 详细信息

```
kubectl describe pods/nginx
kubectl describe pods my-pod
kubectl describe -f pod.json
```

查看 Node 详细信息

```
kubectl describe nodes c1
```

查看 RC 关联的 Pod 信息

```
kubectl describe pods <rc-name>
```

更新修补资源

滚动更新

Plain Text

```
# 滚动更新 pod frontend-v1
kubectl rolling-update frontend-v1 -f frontend-v2.json

# 更新资源名称并更新镜像
kubectl rolling-update frontend-v1 frontend-v2 --image=image:v2

# 更新 frontend pod 中的镜像
kubectl rolling-update frontend --image=image:v2

# 退出已存在的进行中的滚动更新
kubectl rolling-update frontend-v1 frontend-v2 --rollback

# 强制替换；删除后重新创建资源；服务会中断
kubectl replace --force -f ./pod.json

# 添加标签
kubectl label pods my-pod new-label=awesome

# 添加注解
kubectl annotate pods my-pod icon-url=http://goo.gl/XXBTWq
```

修补资源

Plain Text

```
# 部分更新节点
kubectl patch node k8s-node-1 -p '{"spec":{"unschedulable":true}}'

# 更新容器镜像；spec.containers[*].name 是必须的，因为这是合并的关键字
kubectl patch pod valid-pod -p \
    '{"spec":{"containers":[{"name":"kubernetes-serve-hostname","image":"new
image"}]}}'
```

Scale 资源

Plain Text

```
# Scale a replicaset named 'foo' to 3
kubectl scale --replicas=3 rs/foo

# Scale a resource specified in "foo.yaml" to 3
kubectl scale --replicas=3 -f foo.yaml

# If the deployment named mysql's current size is 2, scale mysql to 3
kubectl scale --current-replicas=2 --replicas=3 deployment/mysql

# Scale multiple replication controllers
kubectl scale --replicas=5 rc/foo rc/bar rc/baz
```

删除资源对象

基于 xxx.yaml 文件删除 Pod 对象

Plain Text

```
# yaml文件名字按照你创建时的文件一致
kubectl delete -f xxx.yaml
```

删除包括某个 label 的 pod 对象

Plain Text

```
kubectl delete pods -l name=<label-name>
```

删除包括某个 label 的 service 对象

Plain Text

```
kubectl delete services -l name=<label-name>
```

删除包括某个 label 的 pod 和 service 对象

Plain Text

```
kubectl delete pods,services -l name=<label-name>
```

删除所有 pod/services 对象

Plain Text

```
kubectl delete pods --all
kubectl delete service --all
kubectl delete deployment --all
```

编辑资源文件

在编辑器中编辑任何 API 资源

Plain Text

```
# 编辑名为docker-registry的service
kubectl edit svc/docker-registry
```

直接执行命令

在寄主机上，不进入容器直接执行命令

执行 pod 的 date 命令，默认使用 pod 的第一个容器执行

Plain Text

```
kubectl exec mypod -- date
kubectl exec mypod --namespace=test -- date
```

指定 pod 中某个容器执行 date 命令

Plain Text

```
kubectl exec mypod -c ruby-container -- date
```

进入某个容器

Plain Text

```
kubectl exec mypod -c ruby-container -it -- bash
```

查看容器日志

直接查看日志

Plain Text

```
# 不实时刷新kubectl logs mypod
kubectl logs mypod --namespace=test
```

查看日志实时刷新

Plain Text

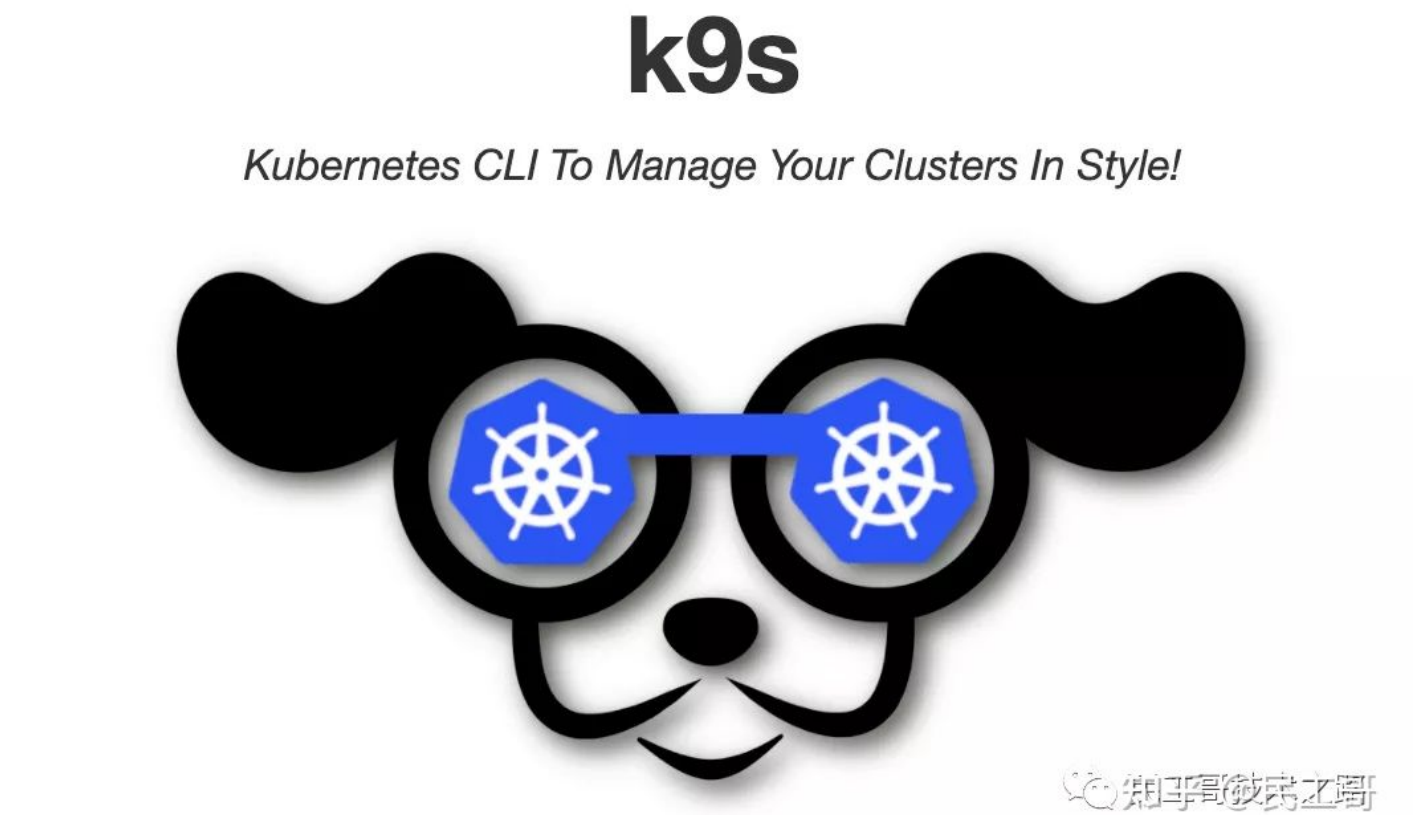
```
kubectl logs -f mypod -c ruby-container
```

管理工具

Kubernetes正在不断加快在云原生环境的应用，但如何以统一、安全的方式对运行于任何地方的Kubernetes集群进行管理面临着挑战，而有效的管理工具能够大大降低管理的难度。

K9s

k9s是基于终端的资源仪表板。它只有一个命令行界面。无论在Kubernetes仪表板Web UI上做什么，都可以在终端使用K9s仪表板工具进行相同的操作。k9s持续关注Kubernetes集群，并提供命令以使用集群上定义的资源。



详细介绍：Kubernetes 集群管理工具 K9S

推荐：轻松管理 Kubernetes 集群的7个工具

生产环境最佳实践

使用Kubernetes的一些策略，在安全性、监控、网络、治理、存储、容器生命周期管理和平台选择方面应用最佳实践。下面让我们来看看Kubernetes的一些生产最佳实践。在生产中运行Kubernetes并不容易; 有以下几个方面需要注意。

是否使用存活探针和就绪探针进行健康检查?

管理大型分布式系统可能会很复杂，特别是当出现问题时，我们无法及时得到通知。为了确保应用实例正常工作，设置Kubernetes健康检查至关重要。

通过创建自定义运行健康检查，可以有效避免分布式系统中僵尸服务运行，具体可以根据环境和需要对其进行调整。

就绪探针的目的是让Kubernetes知道该应用是否已经准备好为流量服务。Kubernetes将始终确保准备就绪探针通过之后开始分配服务，将流量发送到Pod。

Liveness-存活探针

你怎么知道你的应用程序是活的还是死的?存活探针可以让你做到这一点。如果你的应用死了，Kubernetes会移除旧的Pod并用新Pod替换它。

Resource Management-资源管理

为单个容器指定资源请求和限制是一个很好的实践。另一个好的实践是将Kubernetes环境划分为不同团队、部门、应用程序和客户机的独立名称空间。

Kubernetes资源使用情况

Kubernetes资源使用指的是容器/pod在生产中所使用的资源数量。

因此，密切关注pods的资源使用情况是非常重要的。一个明显的原因是成本，因为越高的资源利用证明越少的资源浪费。

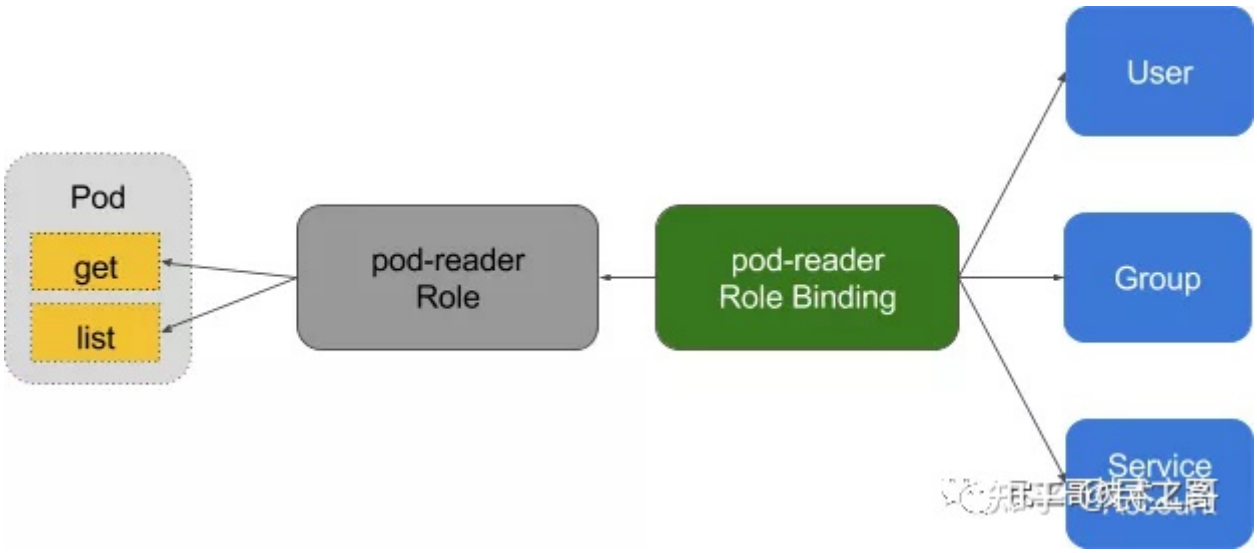
Resource utilization资源利用率

Ops团队通常希望优化和最大化pods消耗的资源百分比。资源使用情况是Kubernetes环境实际优化程度的指标之一。

您可以认为优化后的Kubernetes环境中运行的容器的平均CPU等资源利用率是最优的。

启用RBAC

RBAC代表基于角色的访问控制。它是一种用于限制系统/网络上的用户和应用程序的访问和准入的方法。

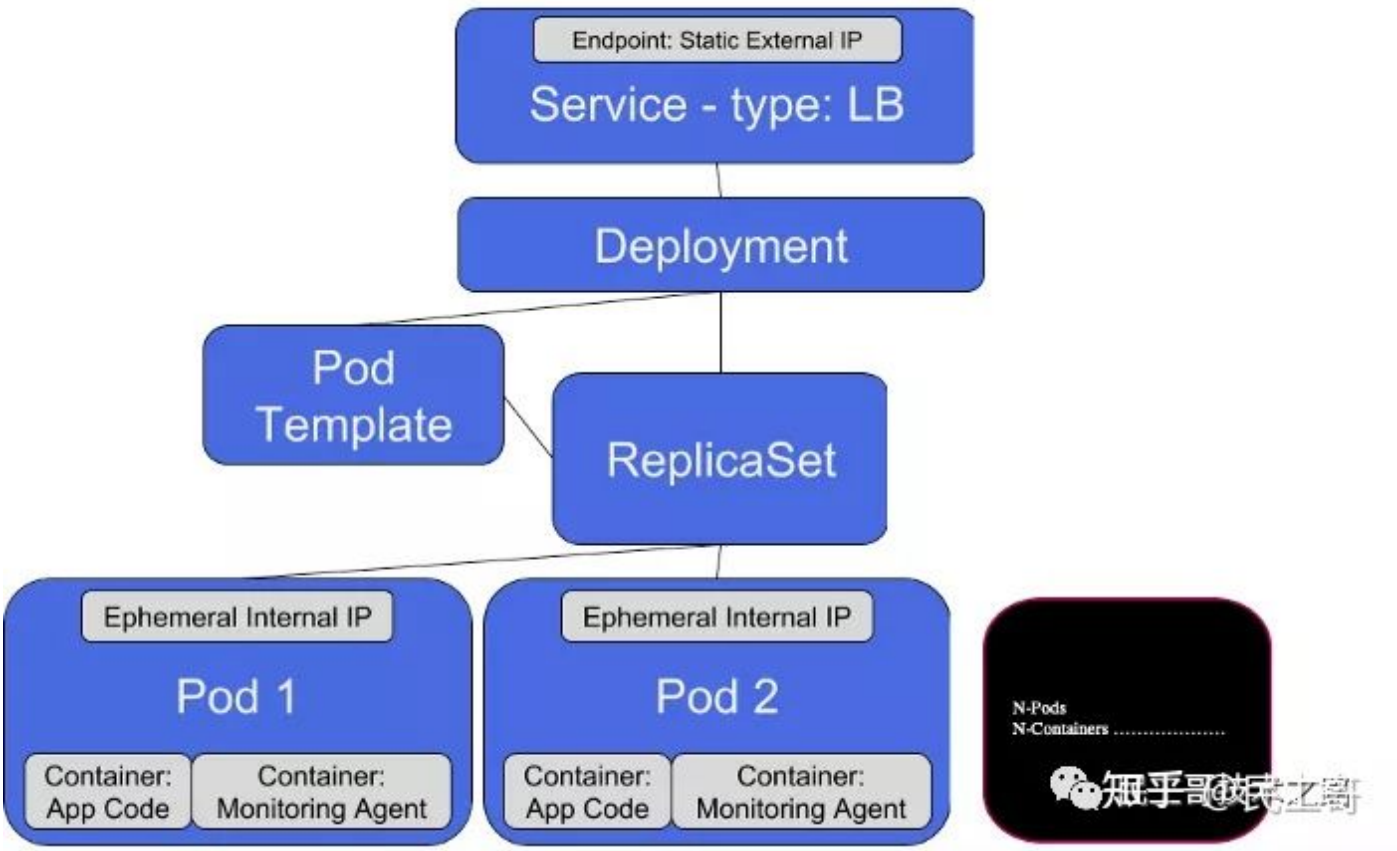


他们从Kubernetes 1.8版本引入了RBAC。使用rbac.authorization.k8s RBAC用于创建授权策略。

在Kubernetes中，RBAC用于授权，使用RBAC，您将能够授予用户、帐户、添加/删除权限、设置规则等权限。因此，它基本上为Kubernetes集群添加了额外的安全层。RBAC限制谁可以访问您的生产环境和集群。

集群置备和负载均衡

生产级Kubernetes基础设施通常需要考虑某些关键方面，例如高可用性、多主机、多etcd Kubernetes集群等。此类集群的配置通常涉及到Terraform或Ansible等工具。



一旦集群都设置好了，并且为运行应用程序创建了pods，这些pods就配备了负载平衡器;这些负载均衡器将流量路由到服务。开源的Kubernetes项目并不是默认的负载平衡器;因此，它需要与NGINX Ingress controller与HAProxy或ELB等工具集成，或任何其他工具，扩大Kubernetes的Ingress插件，以提供负载均衡能力。

给Kubernetes对象添加标签

标签就像附加到对象上的键/值对，比如pods。标签是用来标识对象的属性的，这些属性对用户来说是重要的和有意义的。

在生产中使用Kubernetes时，不能忽视的一个重要问题是标签;标签允许批量查询和操作Kubernetes对象。标签的特殊之处在于，它们还可以用于识别Kubernetes对象并将其组织成组。这样做的最佳用例之一是根据pod所属的应用程序对它们进行分组。在这里，团队可以构建并拥有任意数量的标签约定。

配置网络策略

使用Kubernetes时，设置网络策略至关重要。网络策略只不过是一个对象，它使你能够明确地声明和决定哪些流量是允许的，哪些是不允许的。这样，Kubernetes将能够阻止所有其他不想要的和不符合规则的流量。在我们的集群中定义和限制网络流量是强烈推荐的基本且必要的安全措施之一。

Kubernetes中的每个网络策略都定义了一个如上所述的授权连接列表。无论何时创建任何网络策略，它所引用的所有pod都有资格建立或接受列出的连接。简单地说，网络策略基本上就是授权和允许连接的白名单——一个连接，无论它是到还是从pod，只有在应用于pod的至少一个网络策略允许的情况下才被允许。

集群监控和日志记录

在使用Kubernetes时，监控部署是至关重要的。确保配置、性能和流量保持安全更是重要。如果不进行日志记录和监控，就不可能诊断出发生的问题。为了确保合规性，监视和日志记录变得非常重要。在进行监视时，有必要在体系结构的每一层上设置日志记录功能。生成的日志将帮助我们启用安全工具、审计功能和分析性能。

从无状态应用程序开始

运行无状态应用要比运行有状态应用简单得多，但随着Kubernetes运营商的不断增长，这种想法正在改变。对于刚接触Kubernetes的团队来说，建议首先使用无状态应用程序。

建议使用无状态后端，这样开发团队就可以确保不存在长时间运行的连接，从而增加了扩展的难度。使用无状态，开发人员还可以更有效地、零停机部署应用程序。人们普遍认为，无状态应用程序可以方便地根据业务需要进行迁移和扩展。

边启动自动扩缩容

Kubernetes有三种用于部署的自动伸缩功能:水平pod自动伸缩(HPA)、垂直pod自动伸缩(VPA)和集群自动伸缩。

水平pod autoscaler根据感知到的CPU利用率自动扩展deployment、replicationcontroller, replicaset, statefulset的数量。

Vertical pod autoscaling为CPU和内存请求和限制推荐合适的值，它可以自动更新这些值。

Cluster Autoscaler扩展和缩小工作节点池的大小。它根据当前的利用率调整Kubernetes集群的大小。

控制镜像拉取来源

控制在集群中运行所有容器的镜像源。如果您允许您的Pod从公共资源中拉取镜像，您就不知道其中真正运行的是什么。

如果从受信任的注册表中提取它们，则可以在注册表上应用策略以提取安全和经过认证的镜像。

持续学习

不断评估应用程序的状态和设置，以学习和改进。例如，回顾容器的历史内存使用情况可以得出这样的结论:我们可以分配更少的内存，在长期内节省成本。

保护重要服务

使用Pod优先级，您可以决定设置不同服务运行的重要性。例如，为了更好的稳定性，你需要确保RabbitMQ pod比你的应用pod更重要。或者你的入口控制器pods比数据处理pods更重要，以保持服务对用户可用。

零停机时间

通过在HA中运行所有服务，支持集群和服务的零停机升级。这也将保证您的客户获得更高的可用性。

使用pod反亲和性来确保在不同的节点上调度一个pod的多个副本，从而通过计划中的和计划外的集群节点停机来确保服务可用性。

使用pod Disruptions策略，不惜一切代价确保您有最低的Pod副本数量!

计划失败

硬件最终会失败，软件最终会运行。--（迈克尔·哈顿）

结论

众所周知，Kubernetes实际上已经成为DevOps领域的编排平台标准。Kubernetes从可用性、可伸缩性、安全性、弹性、资源管理和监控的角度来应对生产环境产生的风暴。由于许多公司都在生产中使用Kubernetes，因此必须遵循上面提到的最佳实践，以顺利和可靠地扩展应用程序。

内容来源：https://my.oschina.net/u/1787735/blog/4870582

介绍5 款顶级 Kubernetes 日志监控工具

对于新安装的 Kubernetes，经常出现的一个问题是 Service 没有正常工作。如果您已经运行了 Deployment 并创建了一个 Service，但是当您尝试访问它时没有得到响应，希望这份文档（全网最详细的 K8s Service 不能访问排查流程）能帮助您找出问题所在。

Kubernetes 常见问题总结

如何删除不一致状态下的 rc,deployment,service

在某些情况下,经常发现 kubectl 进程挂起现象,然后在 get 时候发现删了一半,而另外的删除不了

Plain Text

```
[root@k8s-master ~]# kubectl get -f fluentd-elasticsearch/
NAME DESIRED CURRENT READY AGE
rc/elasticsearch-logging-v1 0 2 2 15h

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
deploy/kibana-logging 0 1 1 1 15h
Error from server (NotFound): services "elasticsearch-logging" not found
Error from server (NotFound): daemonsets.extensions "fluentd-es-v1.22" not found
Error from server (NotFound): services "kibana-logging" not found
```

删除这些 deployment,service 或者 rc 命令如下:

Plain Text

```
kubectl delete deployment kibana-logging -n kube-system --cascade=false

kubectl delete deployment kibana-logging -n kube-system  --ignore-not-found

delete rc elasticsearch-logging-v1 -n kube-system --force now --grace-period=0
```

删除不了后如何重置 etcd

Plain Text

```
rm -rf /var/lib/etcd/*
```

删除后重新 reboot master 结点。

Plain Text

```
reset etcd 后需要重新设置网络

etcdctl mk /atomic.io/network/config '{ "Network": "192.168.0.0/16" }'
```

启动 apiserver 失败

每次启动都是报如下问题：

Plain Text

```
start request repeated too quickly for kube-apiserver.service
```

但其实不是启动频率问题,需要查看, /var/log/messages,在我的情况中是因为开启 ServiceAccount 后找不到 ca.crt 等文件,导致启动出错。

Plain Text

```
May 21 07:56:41 k8s-master kube-apiserver: Flag --port has been deprecated, see --insecure-port instead.
May 21 07:56:41 k8s-master kube-apiserver: F0521 07:56:41.692480 4299 universal_validation.go:104] Validate server run options failed: unable to load client CA file: open /var/run/kubernetes/ca.crt: no such file or directory
May 21 07:56:41 k8s-master systemd: kube-apiserver.service: main process exited, code=exited, status=255/n/a
May 21 07:56:41 k8s-master systemd: Failed to start Kubernetes API Server.
May 21 07:56:41 k8s-master systemd: Unit kube-apiserver.service entered failed state.
May 21 07:56:41 k8s-master systemd: kube-apiserver.service failed.
May 21 07:56:41 k8s-master systemd: kube-apiserver.service holdoff time over, scheduling restart.
May 21 07:56:41 k8s-master systemd: start request repeated too quickly for kube-apiserver.service
May 21 07:56:41 k8s-master systemd: Failed to start Kubernetes API Server.
```

在部署 fluentd 等日志组件的时候,很多问题都是因为需要开启 ServiceAccount 选项需要配置安全导致,所以说到底还是需要配置好 ServiceAccount.

出现 Permission denied 情况

在配置 fluentd 时候出现cannot create /var/log/fluentd.log: Permission denied 错误,这是因为没有关掉 SElinux 安全导致。

可以在 /etc/selinux/config 中将 SELINUX=enforcing 设置成 disabled,然后 reboot

基于 ServiceAccount 的配置

首先生成各种需要的 keys,k8s-master 需替换成 master 的主机名.

Plain Text

```
openssl genrsa -out ca.key 2048
openssl req -x509 -new -nodes -key ca.key -subj "/CN=k8s-master" -days 10000 -out ca.crt
openssl genrsa -out server.key 2048

echo subjectAltName=IP:10.254.0.1 > extfile.cnf

#ip由下述命令决定

#kubectl get services --all-namespaces |grep 'default'|grep 'kubernetes'|grep '443'|awk '{print $3}'

openssl req -new -key server.key -subj "/CN=k8s-master" -out server.csr

openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial -extfile extfile.cnf -out server.crt -days 10000
```

如果修改 /etc/kubernetes/apiserver 的配置文件参数的话,通过 systemctl start kube-apiserver 启动失败,出错信息为:

Plain Text

```
Validate server run options failed: unable to load client CA file: open /root/keys/ca.crt: permission denied
```

但可以通过命令行启动 API Server

Plain Text

```
/usr/bin/kube-apiserver --logtostderr=true --v=0 --etcd-servers=http://k8s-master:2379 --address=0.0.0.0 --port=8080 --kubelet-port=10250 --allow-privileged=true --service-cluster-ip-range=10.254.0.0/16 --admission-control=ServiceAccount --insecure-bind-address=0.0.0.0 --client-ca-file=/root/keys/ca.crt --tls-cert-file=/root/keys/server.crt --tls-private-key-file=/root/keys/server.key --basic-auth-file=/root/keys/basic_auth.csv --secure-port=443 &>> /var/log/kubernetes/kube-apiserver.log &
```

命令行启动 Controller-manager

Plain Text

```
/usr/bin/kube-controller-manager --logtostderr=true --v=0 --master=http://k8s-master:8080 --root-ca-file=/root/keys/ca.crt --service-account-private-key-file=/root/keys/server.key & >>/var/log/kubernetes/kube-controller-manage.log
```

ETCD 启动不起来-问题<1>

etcd是kubernetes 集群的zookeeper进程，几乎所有的service都依赖于etcd的启动，比如flanneld,apiserver,docker.....在启动etcd是报错日志如下：

Plain Text

```
May 24 13:39:09 k8s-master systemd: Stopped Flanneld overlay address etcd agent.
May 24 13:39:28 k8s-master systemd: Starting Etcd Server...
May 24 13:39:28 k8s-master etcd: recognized and used environment variable ETCD_ADVERTISE_CLIENT_URLS=http://etcd:2379,http://etcd:4001
May 24 13:39:28 k8s-master etcd: recognized environment variable ETCD_NAME, but unused: shadowed by corresponding flag
May 24 13:39:28 k8s-master etcd: recognized environment variable ETCD_DATA_DIR, but unused: shadowed by corresponding flag
May 24 13:39:28 k8s-master etcd: recognized environment variable ETCD_LISTEN_CLIENT_URLS, but unused: shadowed by corresponding flag
May 24 13:39:28 k8s-master etcd: etcd Version: 3.1.3
May 24 13:39:28 k8s-master etcd: Git SHA: 21fdcc6
May 24 13:39:28 k8s-master etcd: Go Version: go1.7.4
May 24 13:39:28 k8s-master etcd: Go OS/Arch: linux/amd64
May 24 13:39:28 k8s-master etcd: setting maximum number of CPUs to 1, total number of available CPUs is 1
May 24 13:39:28 k8s-master etcd: the server is already initialized as member before, starting as etcd member...
May 24 13:39:28 k8s-master etcd: listening for peers on http://localhost:2380
May 24 13:39:28 k8s-master etcd: listening for client requests on 0.0.0.0:2379
May 24 13:39:28 k8s-master etcd: listening for client requests on 0.0.0.0:4001
May 24 13:39:28 k8s-master etcd: recovered store from snapshot at index 140014
May 24 13:39:28 k8s-master etcd: name = master
May 24 13:39:28 k8s-master etcd: data dir = /var/lib/etcd/default.etcd
May 24 13:39:28 k8s-master etcd: member dir = /var/lib/etcd/default.etcd/member
May 24 13:39:28 k8s-master etcd: heartbeat = 100ms
May 24 13:39:28 k8s-master etcd: election = 1000ms
May 24 13:39:28 k8s-master etcd: snapshot count = 10000
May 24 13:39:28 k8s-master etcd: advertise client URLs = http://etcd:2379,http://etcd:4001
May 24 13:39:28 k8s-master etcd: ignored file 0000000000000001-0000000000012700.wal.broken in wal
```

```
0000000000012700.wal.broken in wal
May 24 13:39:29 k8s-master etcd: restarting member 8e9e05c52164694d in cluster cdf818194e3a8c32 at commit index 148905
May 24 13:39:29 k8s-master etcd: 8e9e05c52164694d became follower at term 12
May 24 13:39:29 k8s-master etcd: newRaft 8e9e05c52164694d [peers: [8e9e05c52164694d], term: 12, commit: 148905, applied: 140014, lastindex: 148905, lastterm: 12]
May 24 13:39:29 k8s-master etcd: enabled capabilities for version 3.1
May 24 13:39:29 k8s-master etcd: added member 8e9e05c52164694d [http://localhost:2380] to cluster cdf818194e3a8c32 from store
May 24 13:39:29 k8s-master etcd: set the cluster version to 3.1 from store
May 24 13:39:29 k8s-master etcd: starting server... [version: 3.1.3, cluster version: 3.1]
May 24 13:39:29 k8s-master etcd: raft save state and entries error: open /var/lib/etcd/default.etcd/member/wal/0.tmp: is a directory
May 24 13:39:29 k8s-master systemd: etcd.service: main process exited, code=exited, status=1/FAILURE
May 24 13:39:29 k8s-master systemd: Failed to start Etcd Server.
May 24 13:39:29 k8s-master systemd: Unit etcd.service entered failed state.
May 24 13:39:29 k8s-master systemd: etcd.service failed.
May 24 13:39:29 k8s-master systemd: etcd.service holdoff time over, scheduling restart.
```

核心语句：

Plain Text

```
raft save state and entries error: open /var/lib/etcd/default.etcd/member/wal/0.tmp: is a directory
```

进入相关目录，删除 0.tmp,然后就可以启动啦!

ETCD启动不起来-超时问题<2>

问题背景：当前部署了 3 个 etcd 节点，突然有一天 3 台集群全部停电宕机了。重新启动之后发现 K8S 集群是可以正常使用的，但是检查了一遍组件之后，发现有一个节点的 etcd 启动不了。

经过一遍探查，发现时间不准确，通过以下命令 ntpdate http://ntp.aliyun.com 重新将时间调整正确，重新启动 etcd，发现还是起不来，报错如下：

Plain Text

```
Mar 05 14:27:15 k8s-node2 etcd[3248]: etcd Version: 3.3.13
Mar 05 14:27:15 k8s-node2 etcd[3248]: Git SHA: 98d3084
Mar 05 14:27:15 k8s-node2 etcd[3248]: Go Version: go1.10.8
Mar 05 14:27:15 k8s-node2 etcd[3248]: Go OS/Arch: linux/amd64
Mar 05 14:27:15 k8s-node2 etcd[3248]: setting maximum number of CPUs to 4, total number of available CPUs is 4
```

```
Mar 05 14:27:15 k8s-node2 etcd[3248]: the server is already initialized as member before, starting as etcd member
...
Mar 05 14:27:15 k8s-node2 etcd[3248]: peerTLS: cert = /opt/etcd/ssl/server.pem, key = /opt/etcd/ssl/server-key.pe
m, ca = , trusted-ca = /opt/etcd/ssl/ca.pem, client-cert-auth = false, crl-file =
Mar 05 14:27:15 k8s-node2 etcd[3248]: listening for peers on https://192.168.25.226:2380
Mar 05 14:27:15 k8s-node2 etcd[3248]: The scheme of client url http://127.0.0.1:2379 is HTTP while peer key/cert
files are presented. Ignored key/cert files.
Mar 05 14:27:15 k8s-node2 etcd[3248]: listening for client requests on 127.0.0.1:2379
Mar 05 14:27:15 k8s-node2 etcd[3248]: listening for client requests on 192.168.25.226:2379
Mar 05 14:27:15 k8s-node2 etcd[3248]: member 9c166b8b7cb6ecb8 has already been bootstrapped
Mar 05 14:27:15 k8s-node2 systemd[1]: etcd.service: main process exited, code=exited, status=1/FAILURE
Mar 05 14:27:15 k8s-node2 systemd[1]: Failed to start Etcd Server.
Mar 05 14:27:15 k8s-node2 systemd[1]: Unit etcd.service entered failed state.
Mar 05 14:27:15 k8s-node2 systemd[1]: etcd.service failed.
Mar 05 14:27:15 k8s-node2 systemd[1]: etcd.service failed.
Mar 05 14:27:15 k8s-node2 systemd[1]: etcd.service holdoff time over, scheduling restart.
Mar 05 14:27:15 k8s-node2 systemd[1]: Starting Etcd Server...
Mar 05 14:27:15 k8s-node2 etcd[3258]: recognized environment variable ETCD_NAME, but unused: shadowed by correspo
nding flag
Mar 05 14:27:15 k8s-node2 etcd[3258]: recognized environment variable ETCD_DATA_DIR, but unused: shadowed by corr
esponding flag
Mar 05 14:27:15 k8s-node2 etcd[3258]: recognized environment variable ETCD_LISTEN_PEER_URLS, but unused: shadowed
 by corresponding flag
Mar 05 14:27:15 k8s-node2 etcd[3258]: recognized environment variable ETCD_LISTEN_CLIENT_URLS, but unused: shadow
ed by corresponding flag
Mar 05 14:27:15 k8s-node2 etcd[3258]: recognized environment variable ETCD_INITIAL_ADVERTISE_PEER_URLS, but unuse
d: shadowed by corresponding flag
Mar 05 14:27:15 k8s-node2 etcd[3258]: recognized environment variable ETCD_ADVERTISE_CLIENT_URLS, but unused: sha
dowed by corresponding flag
Mar 05 14:27:15 k8s-node2 etcd[3258]: recognized environment variable ETCD_INITIAL_CLUSTER, but unused: shadowed
by corresponding flag
```

```
36  by corresponding flag
37  Mar 05 14:27:15 k8s-node2 etcd[3258]: recognized environment variable
    ETCD_INITIAL_CLUSTER_TOKEN, but unused: sha
38  dowed by corresponding flag
39  Mar 05 14:27:15 k8s-node2 etcd[3258]: recognized environment variable
    ETCD_INITIAL_CLUSTER_STATE, but unused: sha
40  dowed by corresponding flag
```

解决方法：

检查日志发现并没有特别明显的错误，根据经验来讲，etcd 节点坏掉一个其实对集群没有大的影响，这时集群已经可以正常使用了，但是这个坏掉的 etcd 节点并没有启动，解决方法如下：

进入 etcd 的数据存储目录进行备份 备份原有数据：

Plain Text

```
1  cd /var/lib/etcd/default.etcd/member/
2  cp *  /data/bak/
```

删除这个目录下的所有数据文件

Plain Text

```
1  rm -rf /var/lib/etcd/default.etcd/member/*
```

停止另外两台 etcd 节点，因为 etcd 节点启动时需要所有节点一起启动，启动成功后即可使用。

Plain Text

```
1  #master 节点
2  systemctl stop etcd
3  systemctl restart etcd
4
5  #node1 节点
6  systemctl stop etcd
7  systemctl restart etcd
8
9  #node2 节点
10 systemctl stop etcd
11 systemctl restart etcd
```

CentOS下配置主机互信

在每台服务器需要建立主机互信的用户名执行以下命令生成公钥/密钥，默认回车即可

Plain Text

```
ssh-keygen -t rsa
```

可以看到生成个公钥的文件。

互传公钥，第一次需要输入密码，之后就OK了。

Plain Text

```
ssh-copy-id -i /root/.ssh/id_rsa.pub root@192.168.199.132 (-p 2222)
```

-p 端口 默认端口不加-p，如果更改过端口，就得加上-p. 可以看到是在.ssh/下生成了个authorized_keys的文件，记录了能登陆这台服务器的其他服务器的公钥。

测试看是否能登陆：

Plain Text

```
ssh 192.168.199.132 (-p 2222)
```

CentOS 主机名的修改

Plain Text

```
hostnamectl set-hostname k8s-master1
```

Virtualbox 实现 CentOS 复制和粘贴功能

如果不安装或者不输出，可以将 update 修改成 install 再运行。

Plain Text

```
yum install update
yum update kernel
yum update kernel-devel
yum install kernel-headers
yum install gcc
yum install gcc make
```

运行完后

```
Plain Text
1  sh VBoxLinuxAdditions.run
```

删除Pod一直处于Terminating状态

可以通过下面命令强制删除

```
Plain Text
1  kubectl delete pod NAME --grace-period=0 --force
```

删除namespace一直处于Terminating状态

可以通过以下脚本强制删除

```
Plain Text
1  [root@k8s-master1 k8s]# cat delete-ns.sh
2  #!/bin/bash
3  set -e
4
5  useage(){
6      echo "useage:"
7      echo " delns.sh NAMESPACE"
8  }
9
10 if [ $# -lt 1 ];then
11     useage
12     exit
13 fi
14
15 NAMESPACE=$1
16 JSONFILE=${NAMESPACE}.json
17 kubectl get ns "${NAMESPACE}" -o json > "${JSONFILE}"
18 vi "${JSONFILE}"
19 curl -k -H "Content-Type: application/json" -X PUT --data-binary @"${JSONFLE}"
   \
20     http://127.0.0.1:8001/api/v1/namespaces/"${NAMESPACE}"/finalize
```

容器包含有效的 CPU/内存 requests 且没有指定 limits 可能会出现什么问题?

下面我们创建一个对应的容器，该容器只有 requests 设定，但是没有 limits 设定，

Plain Text

```
- name: busybox-cnt02
    image: busybox
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo hello from cnt02; sleep 10;done"]
    resources:
      requests:
        memory: "100Mi"
        cpu: "100m"
```

这个容器创建出来会有什么问题呢?

其实对于正常的环境来说没有什么问题，但是对于资源型 pod 来说，如果有的容器没有设定 limit 限制，资源会被其他的 pod 抢占走，可能会造成容器应用失败的情况。可以通过 limitrange 策略来去匹配，让 pod 自动设定，前提是要提前配置好limitrange 规则。

来源：https://www.cnblogs.com/passzhang

Kubernetes 上对应用程序进行故障排除的 6 个技巧 推荐给大家，日常排错必备。分享一份阿里云内部超全K8s实战手册，免费下载！

面试题

一个目标：容器操作；两地三中心；四层服务发现；五种 Pod 共享资源；六个 CNI 常用插件；七层负载均衡；八种隔离维度；九个网络模型原则；十类 IP 地址；百级产品线；千级物理机；万级容器；相如无亿，k8s 有亿：亿级日服务人次。

一个目标：容器操作

Kubernetes（k8s）是自动化容器操作的开源平台。这些容器操作包括：部署、调度和节点集群间扩展。

具体功能：

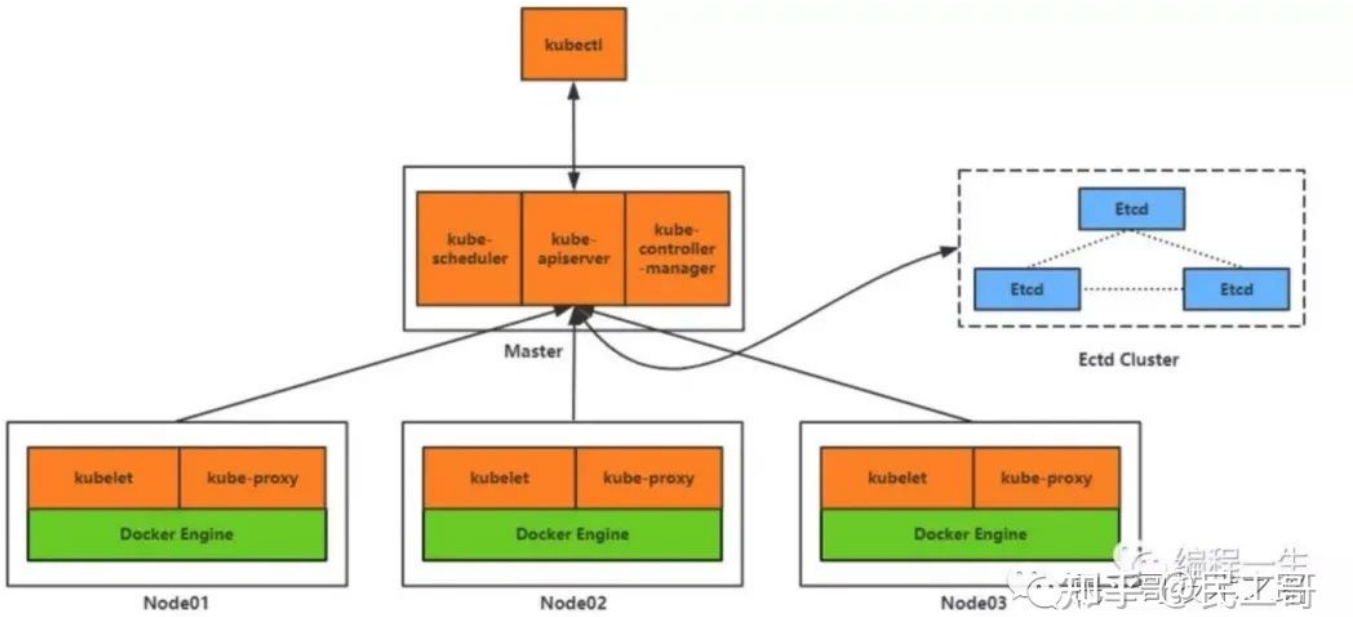
- 自动化容器部署和复制。
- 实时弹性收缩容器规模。
- 容器编排成组，并提供容器间的负载均衡。
- 调度：容器在哪个机器上运行。

组成：

- kubectl：客户端命令行工具，作为整个系统的操作入口。
- kube-apiserver：以 REST API 服务形式提供接口，作为整个系统的控制入口。

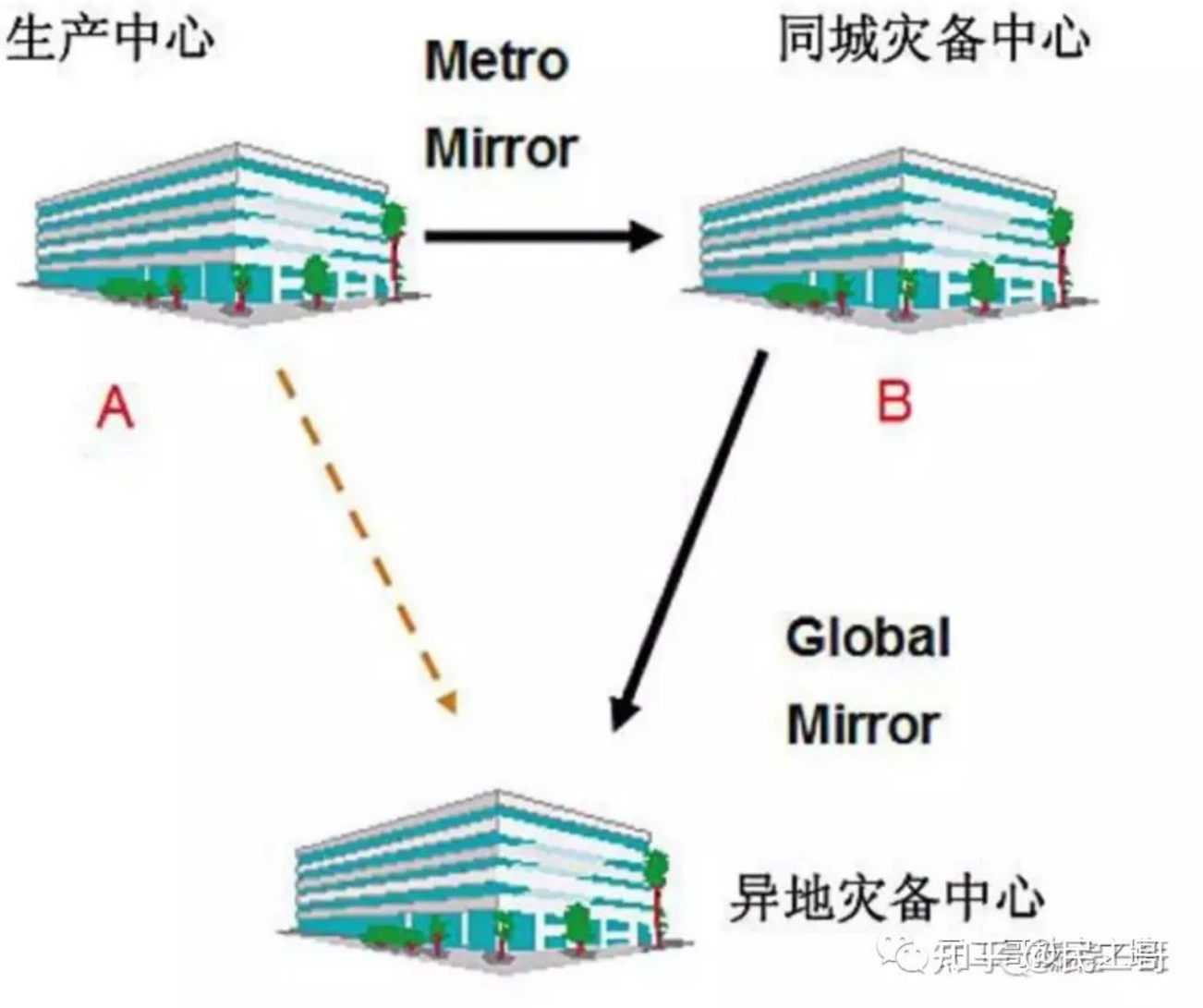
- kube-controller-manager：执行整个系统的后台任务，包括节点状态状况、Pod 个数、Pods 和 Service 的关联等。
- kube-scheduler：负责节点资源管理，接收来自 kube-apiserver 创建 Pods 任务，并分配到某个节点。
- etcd：负责节点间的服务发现和配置共享。
- kube-proxy：运行在每个计算节点上，负责 Pod 网络代理。定时从 etcd 获取到 service 信息来做相应的策略。
- kubelet：运行在每个计算节点上，作为 agent，接收分配该节点的 Pods 任务及管理容器，周期性获取容器状态，反馈给 kube-apiserver。
- DNS：一个可选的 DNS 服务，用于为每个 Service 对象创建 DNS 记录，这样所有的 Pod 就可以通过 DNS 访问服务了。

下面是 k8s 的架构拓扑图：



两地三中心

两地三中心包括本地生产中心、本地灾备中心、异地灾备中心。



两地三中心要解决的一个重要问题就是数据一致性问题。

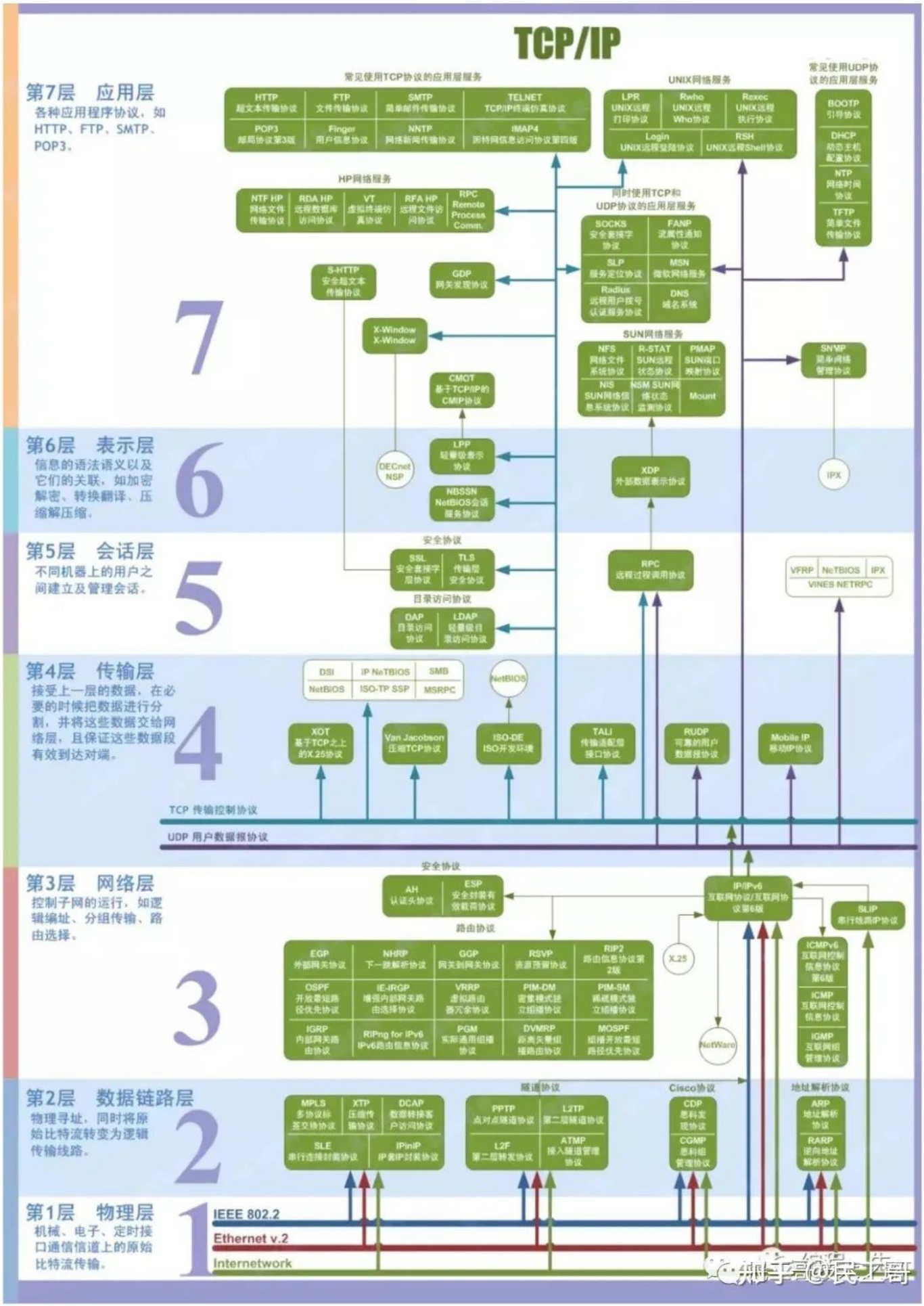
k8s 使用 etcd 组件作为一个高可用、强一致性的服务发现存储仓库。用于配置共享和服务发现。

它作为一个受到 Zookeeper 和 doozer 启发而催生的项目。除了拥有他们的所有功能之外，还拥有以下 4 个特点：

- 简单：基于 HTTP+JSON 的 API 让你用 curl 命令就可以轻松使用。
- 安全：可选 SSL 客户认证机制。
- 快速：每个实例每秒支持一千次写操作。
- 可信：使用 Raft 算法充分实现了分布式。

四层服务发现

先一张图解释一下网络七层协议：



k8s 提供了两种方式进行服务发现：

- 环境变量：当创建一个 Pod 的时候，kubelet 会在该 Pod 中注入集群内所有 Service 的相关环境变量。需要注意的是，要想一个 Pod 中注入某个 Service 的环境变量，则必须 Service 要先比该 Pod 创建。这一点，几乎使得这种方式进行服务发现不可用。 比如，一个 ServiceName 为 redis-master 的 Service，对应的 ClusterIP:Port 为 10.0.0.11:6379，则对应的环境变量为：

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

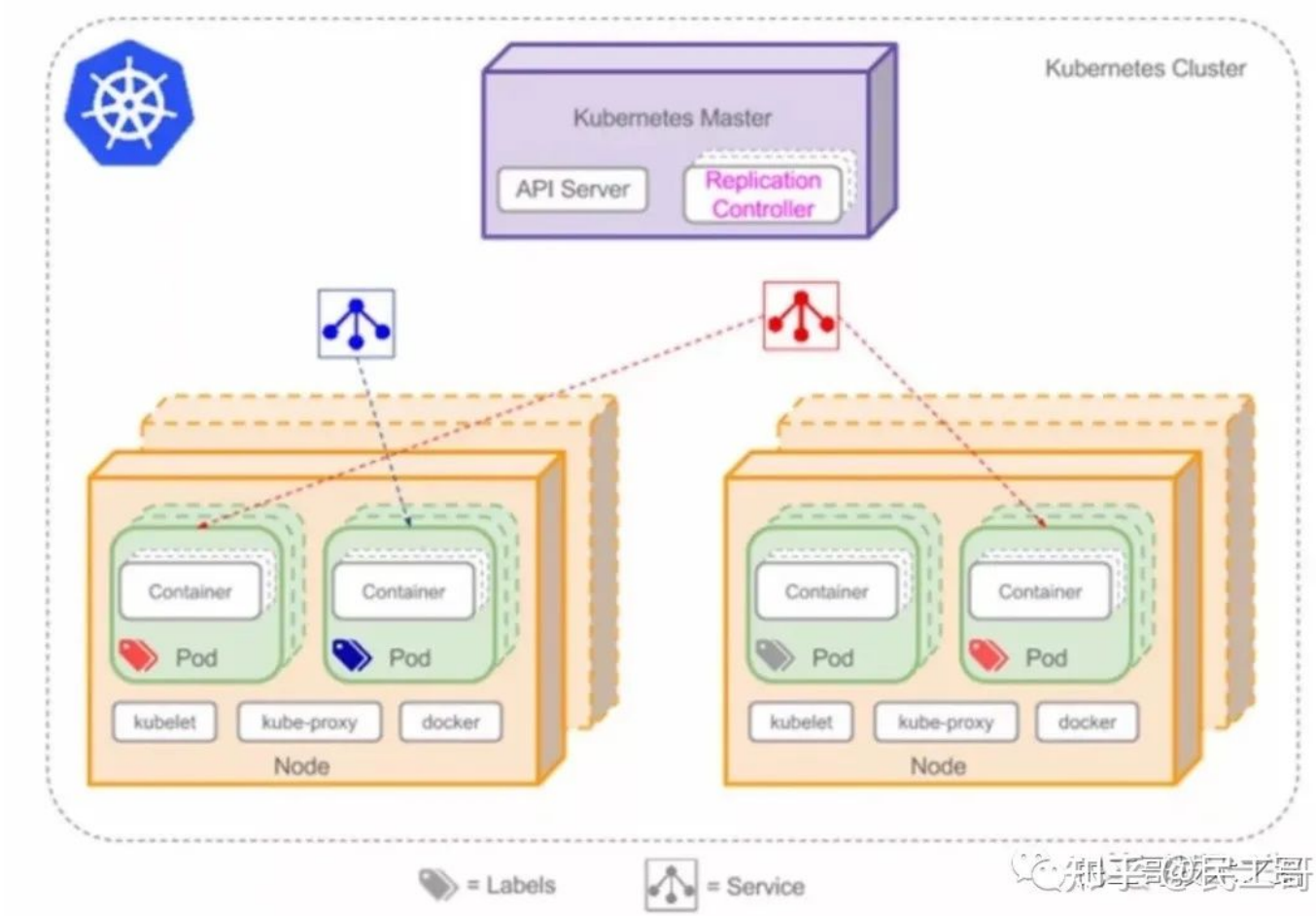
- DNS：可以通过 cluster add-on 的方式轻松的创建 KubeDNS 来对集群内的 Service 进行服务发现。

以上两种方式，一个是基于 TCP，DNS 基于 UDP，它们都是建立在四层协议之上。

五种 Pod 共享资源

Pod 是 k8s 最基本的操作单元，包含一个或多个紧密相关的容器。

一个 Pod 可以被一个容器化的环境看作应用层的“逻辑宿主机”；一个 Pod 中的多个容器应用通常是紧密耦合的，Pod 在 Node 上被创建、启动或者销毁；每个 Pod 里运行着一个特殊的被称之为 Volume 挂载卷，因此他们之间通信和数据交换更为高效。在设计时我们可以充分利用这一特性将一组密切相关的服务进程放入同一个 Pod 中。



同一个 Pod 里的容器之间仅需通过 localhost 就能互相通信。

一个 Pod 中的应用容器共享五种资源：

- PID 命名空间：Pod 中的不同应用程序可以看到其他应用程序的进程 ID。
- 网络命名空间：Pod 中的多个容器能够访问同一个IP和端口范围。
- IPC 命名空间：Pod 中的多个容器能够使用 SystemV IPC 或 POSIX 消息队列进行通信。
- UTS 命名空间：Pod 中的多个容器共享一个主机名。
- Volumes（共享存储卷）：Pod 中的各个容器可以访问在 Pod 级别定义的 Volumes。

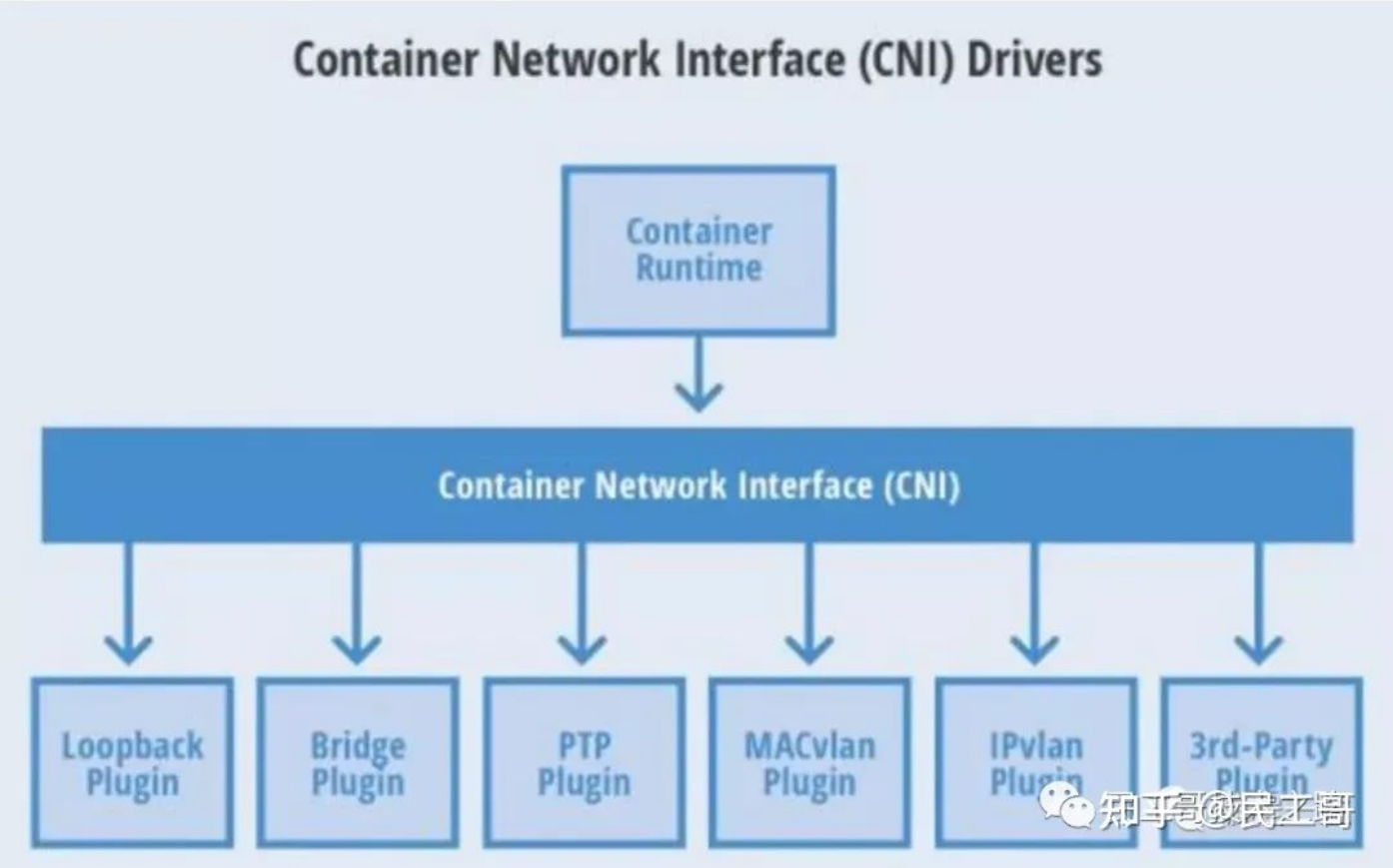
Pod 的生命周期通过 Replication Controller 来管理；通过模板进行定义，然后分配到一个 Node 上运行，在 Pod 所包含容器运行结束后，Pod 结束。

Kubernetes 为 Pod 设计了一套独特的网络配置，包括为每个 Pod 分配一个IP地址，使用 Pod 名作为容器间通信的主机名等。

六个 CNI 常用插件

CNI（Container Network Interface）容器网络接口是 Linux 容器网络配置的一组标准和库，用户需要根据这些标准和库来开发自己的容器网络插件。CNI 只专注解决容器网络连接和容器销毁时的资源释放，提供一套框架。所以 CNI 可以支持大量不同的网络模式，并且容易实现。

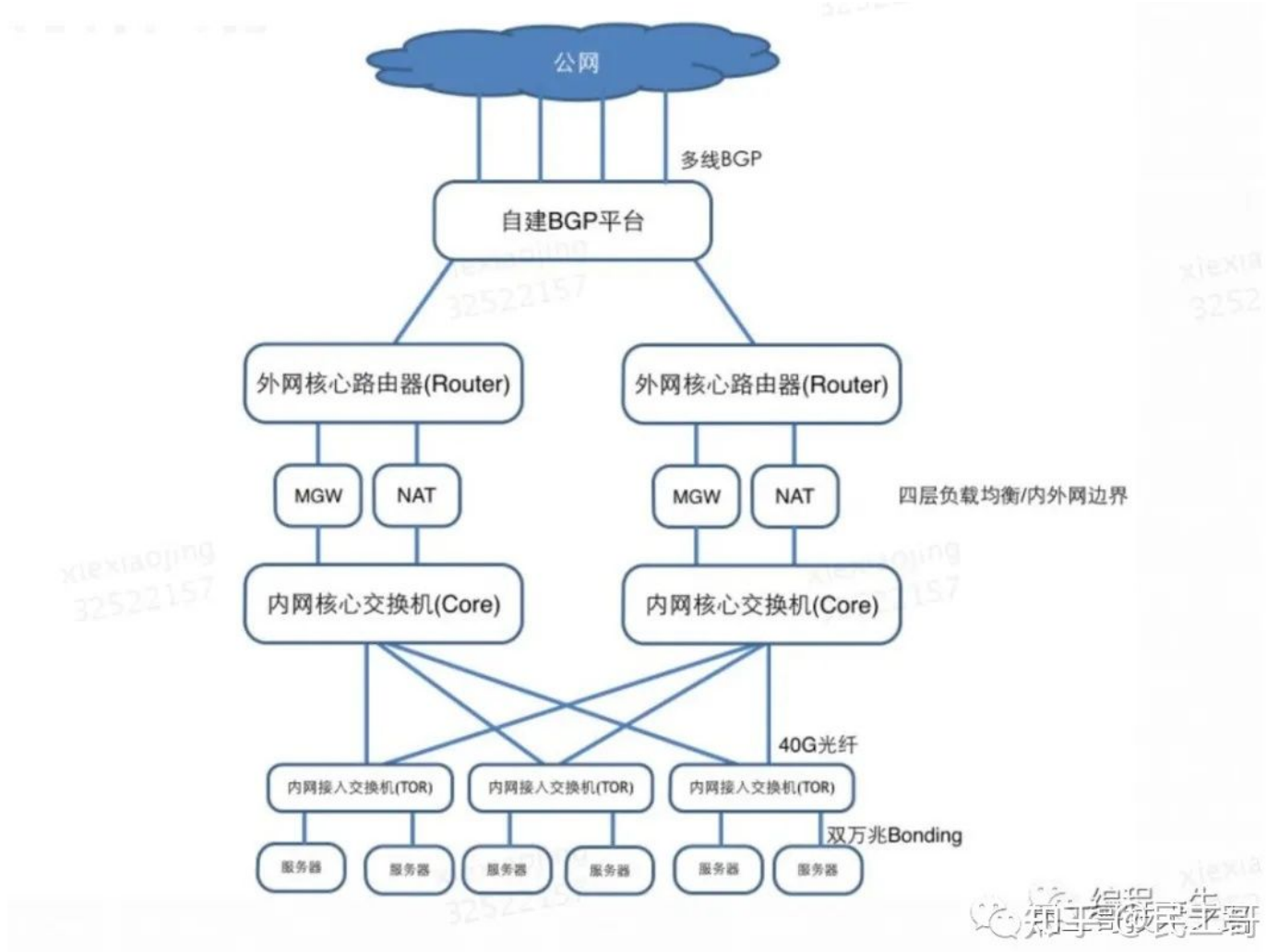
下面用一张图表示六个 CNI 常用插件：



七层负载均衡

提负载均衡就不得不先提服务器之间的通信。

IDC（Internet Data Center）也可称数据中心、机房，用来放置服务器。IDC 网络是服务器间通信的桥梁。



上图里画了很多网络设备，它们都是干啥用的呢？

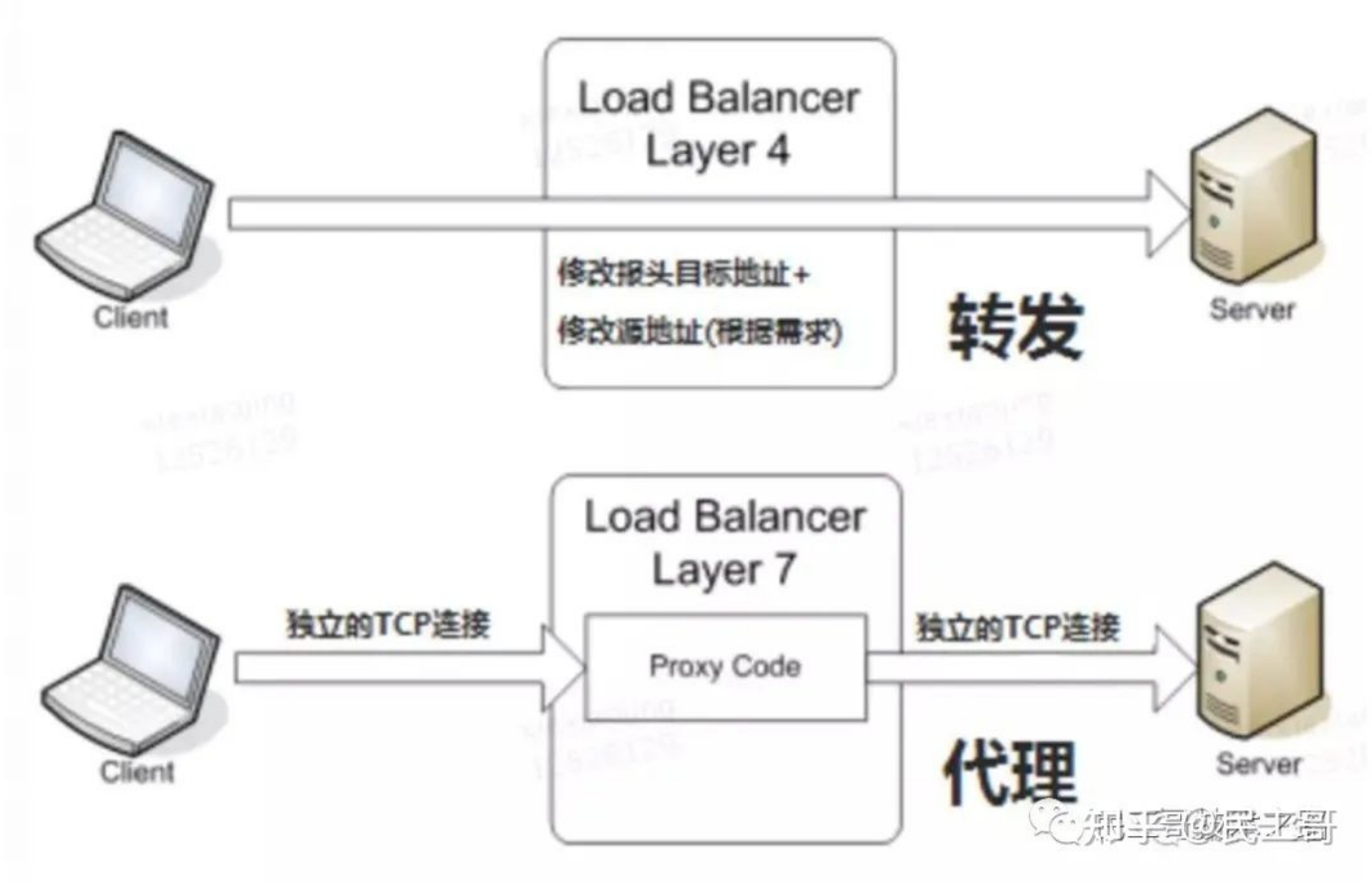
路由器、交换机、MGW/NAT 都是网络设备，按照性能、内外网划分不同的角色。

- 内网接入交换机：也称为 TOR（top of rack），是服务器接入网络的设备。每台内网接入交换机下联 40-48 台服务器，使用一个掩码为 /24 的网段作为服务器内网网段。
- 内网核心交换机：负责 IDC 内各内网接入交换机的流量转发及跨 IDC 流量转发。
- MGW/NAT：MGW 即 LVS 用来做负载均衡，NAT 用于内网设备访问外网时做地址转换。
- 外网核心路由器：通过静态互联运营商或 BGP 互联美团统一外网平台。

先说说各层负载均衡：

- 二层负载均衡：基于 MAC 地址的二层负载均衡。
- 三层负载均衡：基于 IP 地址的负载均衡。
- 四层负载均衡：基于 IP+端口 的负载均衡。
- 七层负载均衡：基于 URL 等应用层信息的负载均衡。

这里用一张图来说说四层和七层负载均衡的区别：



上面四层服务发现讲的主要是 k8s 原生的 kube-proxy 方式。k8s 关于服务的暴露主要是通过 NodePort 方式，通过绑定 minion 主机的某个端口，然后进行 Pod 的请求转发和负载均衡，但这种方式有下面的缺陷：

- Service 可能有很多个，如果每个都绑定一个 Node 主机端口的话，主机需要开放外围的端口进行服务调用，管理混乱。
- 无法应用很多公司要求的防火墙规则。

理想的方式是通过一个外部的负载均衡器，绑定固定的端口，比如 80；然后根据域名或者服务名向后面的 Service IP 转发。

Nginx 很好的解决了这个需求，但问题是如果有的新的服务加入，如何去修改并且加载这些 Nginx 配置？

Kubernetes 给出的方案就是 Ingress。这是一个基于七层的方案。

八种隔离维度

k8s 集群调度这边需要对上面从上到下、从粗粒度到细粒度的隔离做相应的调度策略。

九个网络模型原则

k8s 网络模型要符合四个基础原则、三个网络要求原则、一个架构原则、一个 IP 原则。

每个 Pod 都拥有一个独立的 IP 地址，而且假定所有 Pod 都在一个可以直接连通的、扁平的网络空间中，不管是否运行在同一 Node 上都可以通过 Pod 的 IP 来访问。

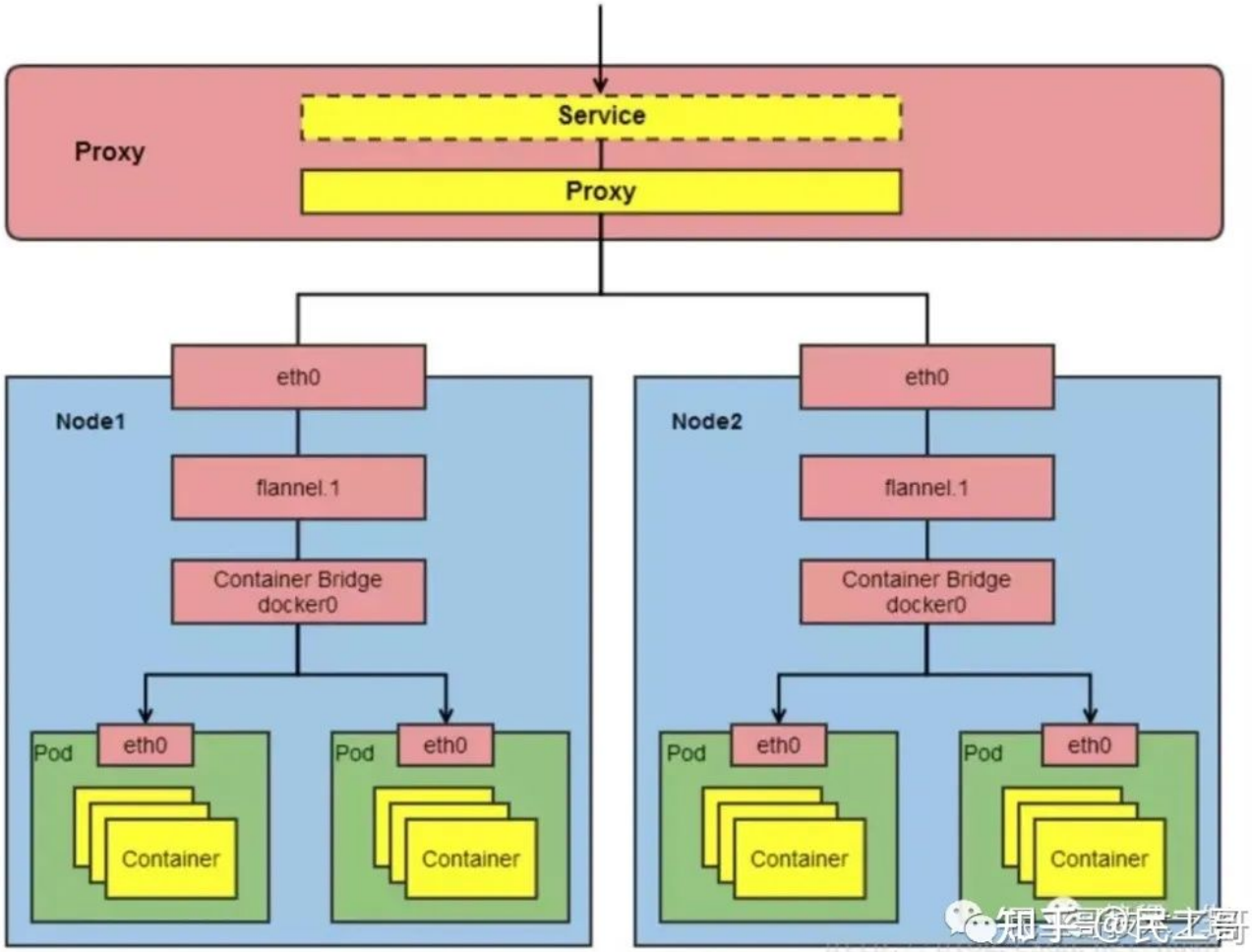
k8s 中的 Pod 的 IP 是最小粒度 IP。同一个 Pod 内所有的容器共享一个网络堆栈，该模型称为 IP-per-Pod 模型。

- Pod 由 docker0 实际分配的 IP。
- Pod 内部看到的 IP 地址和端口与外部保持一致。
- 同一个 Pod 内的不同容器共享网络，可以通过localhost来访问对方的端口，类似同一个虚拟机内不同的进程。

IP-per-Pod 模型从端口分配、域名解析、服务发现、负载均衡、应用配置等角度看，Pod 可以看做是一台独立的虚拟机或物理机。

- 所有容器都可以不用 NAT 的方式同别的容器通信。
- 所有节点都可以在不同 NAT 方式下同所有容器通信，反之亦然。
- 容器的地址和别人看到的地址是同一个地址。

要符合下面的架构：



由上图架构引申出来 IP 概念从集群外部到集群内部：

图片导入失败，请重新上传

图片

十类IP地址

大家都知道 IP 地址分为 ABCDE 类，另外还有五类特殊用途的 IP。

第一类

Plain Text

```
A 类：1.0.0.0-1226.255.255.255，默认子网掩码/8，即255.0.0.0。
B 类：128.0.0.0-191.255.255.255，默认子网掩码/16，即255.255.0.0。
C 类：192.0.0.0-223.255.255.255，默认子网掩码/24，即255.255.255.0。
D 类：224.0.0.0-239.255.255.255，一般用于组播。
E 类：240.0.0.0-255.255.255.255(其中255.255.255.255为全网广播地址)。E 类地址一般用于研究用途。
```

第二类

Plain Text

```
0.0.0.0
严格来说，0.0.0.0 已经不是一个真正意义上的 IP 地址了。它表示的是这样一个集合：所有不清楚的主机和目的网络。这里的不清楚是指在本机的路由表里没有特定条目指明如何到达。作为缺省路由。
127.0.0.1 本机地址。
```

第三类

Plain Text

```
224.0.0.1 组播地址。
如果你的主机开启了IRDP（internet路由发现，使用组播功能），那么你的主机路由表中应该有这样一条路由。
```

第四类

Plain Text

```
169.254.x.x
使用了 DHCP 功能自动获取了 IP 的主机，DHCP 服务器发生故障，或响应时间太长而超出了一个系统规定的时间，系统会为你分配这样一个 IP，代表网络不能正常运行。
```

第五类

Plain Text

```
10.xxx、172.16.x.x~172.31.x.x、192.168.x.x 私有地址。
大量用于企业内部。保留这样的地址是为了避免亦或是哪个接入公网时引起地址混乱。
```

链接：http://blog.csdn.net/huakai_sun/article/details/82378856