



# Lecture 12: 云原生

SSE316: 云计算技术  
Cloud Computing Technologies

---

陈壮彬

软件工程学院

chenzhb36@mail.sysu.edu.cn

# Today's topics

□ 云原生的概念

□ 云原生的代表技术

□ 云原生应用的 12 要素

□ Kubernetes

# 云原生 Cloud Native

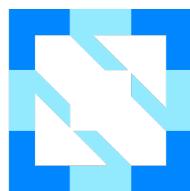
- “我们最近在搞云原生应用”
- “我们最近把业务迁移到云原生平台了”
- “我们的产品是基于云原生技术构建的”
- “云原生让我们的工作变得更简单，更快交付产品”



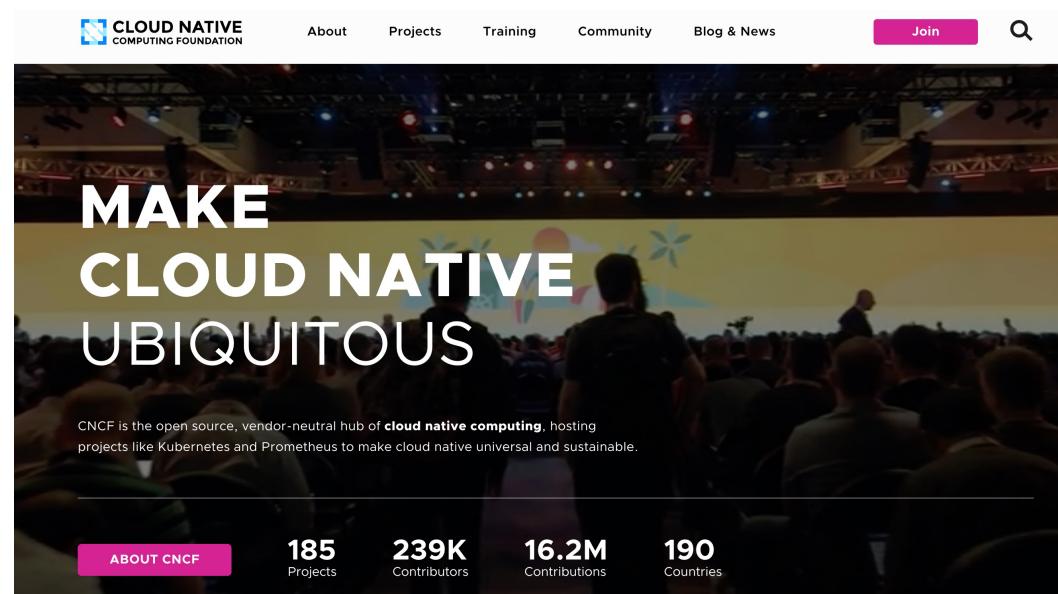
# CNCF Cloud Native Definition v1.1

云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用

云原生技术使工程师能够轻松地对系统作出频繁和可预测的重大变更



**CLOUD NATIVE  
COMPUTING FOUNDATION**



The screenshot shows the CNCF homepage. At the top, there's a navigation bar with links for About, Projects, Training, Community, Blog & News, Join, and a search icon. Below the navigation is a large banner with the text "MAKE CLOUD NATIVE UBIQUITOUS" overlaid on a background image of a conference stage. A subtitle below the banner reads: "CNCF is the open source, vendor-neutral hub of **cloud native computing**, hosting projects like Kubernetes and Prometheus to make cloud native universal and sustainable." At the bottom of the page, there are four statistics: "185 Projects", "239K Contributors", "16.2M Contributions", and "190 Countries".

<https://www.cncf.io/>

<https://github.com/cncf/toc/blob/main/DEFINITION.md>

**Native**  
**本地的、原生的**

**Cloud Native**  
“为云环境或平台原生设计和优化”

# 云原生 Cloud Native

2015 年，Pivotal 公司的马特斯泰恩 (Matt Stine) 提出 Cloud Native 这一概念

**云原生的主旨是构建运行在云端的应用程序，  
致力于使应用程序能够最大限度地  
利用云计算技术的特性和优势**

## CLOUD NATIVE COMPUTING



# 其他技术领域的“Native”

□ Web Native：利用 Web 环境的特性和优势

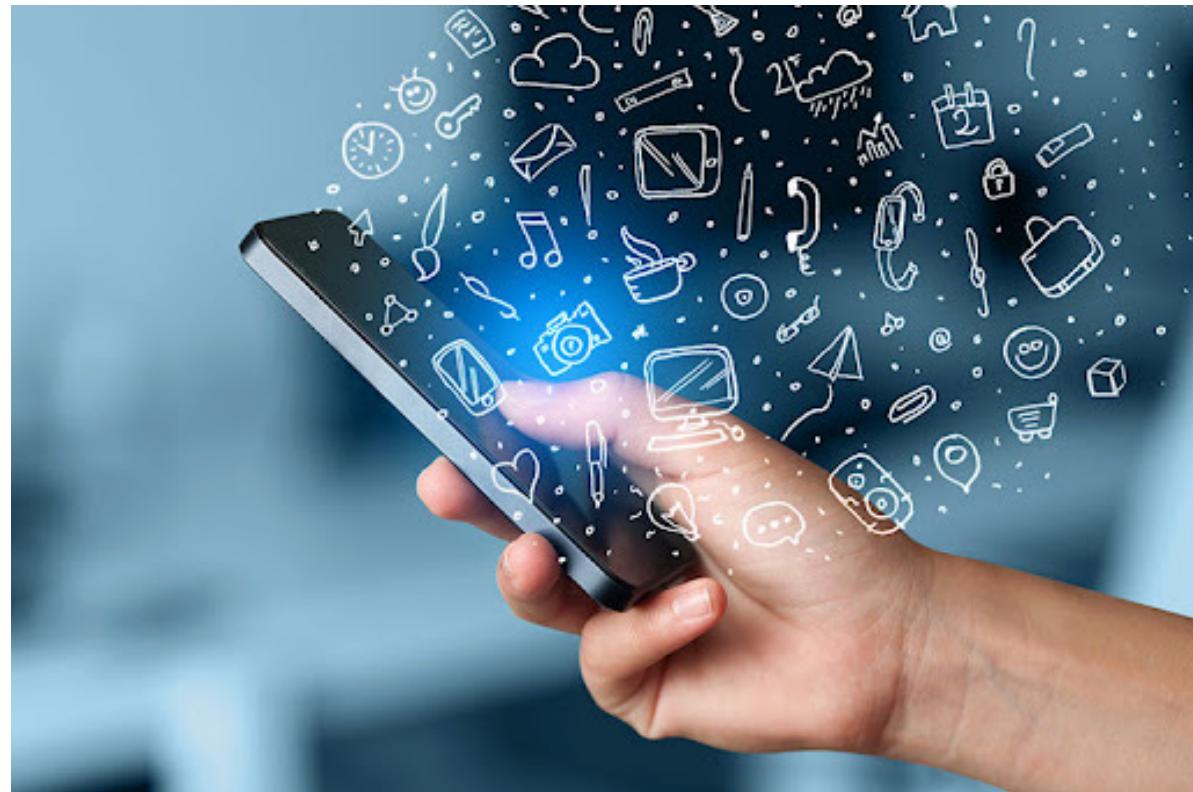
- 超文本链接
- 浏览器兼容性
- JavaScript
- ...



# 其他技术领域的“Native”

□ Mobile Native：为移动设备原生设计和优化的应用

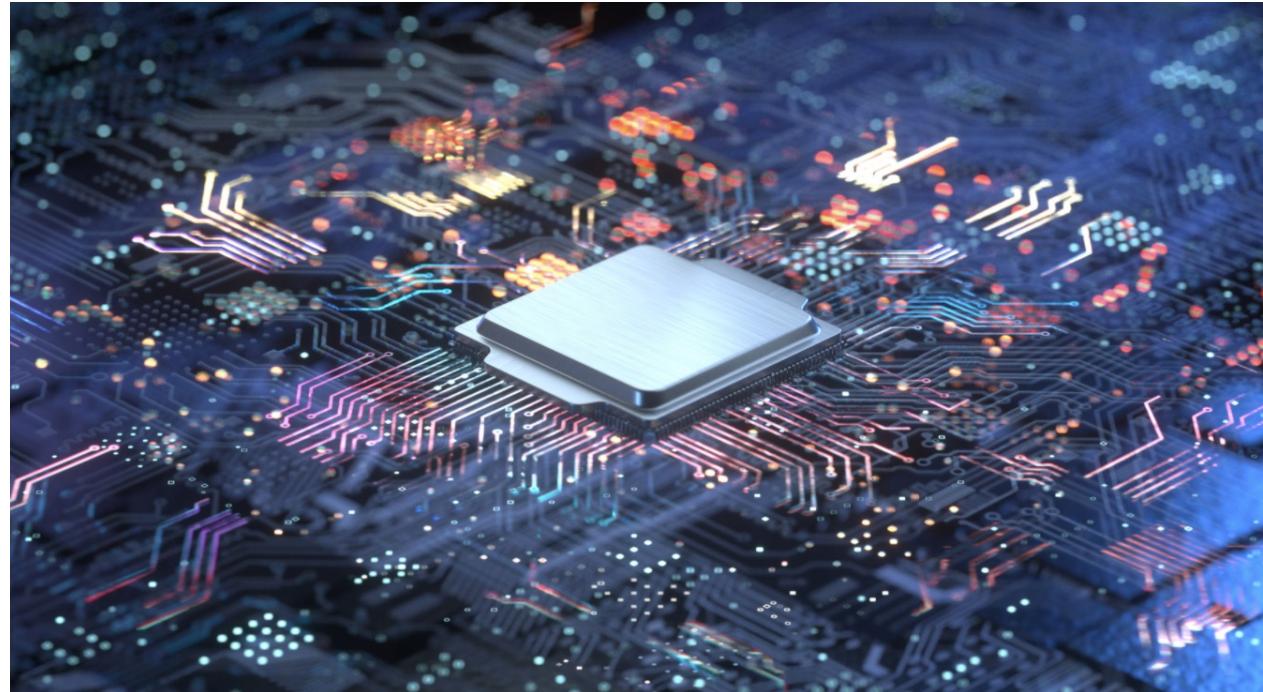
- 触摸屏
- GPS
- 加速度计
- ...



# 其他技术领域的“Native”

□ Hardware Native：为特定硬件平台设计和优化的软件

- 特定的 CPU 指令集
- 硬件加速功能
- ...



# 云原生 vs. 云计算 – 相同点

都是现代  
IT 的核心

云计算和云原生都是现代 IT 架构的重要组成部分，强调灵活性、可扩展性和效率

弹性

云计算和云原生都强调弹性，即能否根据需求快速伸缩资源

# 云原生 vs. 云计算 – 不同点

目标不同

云计算主要关注提供计算资源；云原生关注如何最好地利用这些资源来构建和运行应用

技术和实践不同

云计算主要涉及基础设施的管理和优化；云原生涉及微服务、容器、持续交付等技术

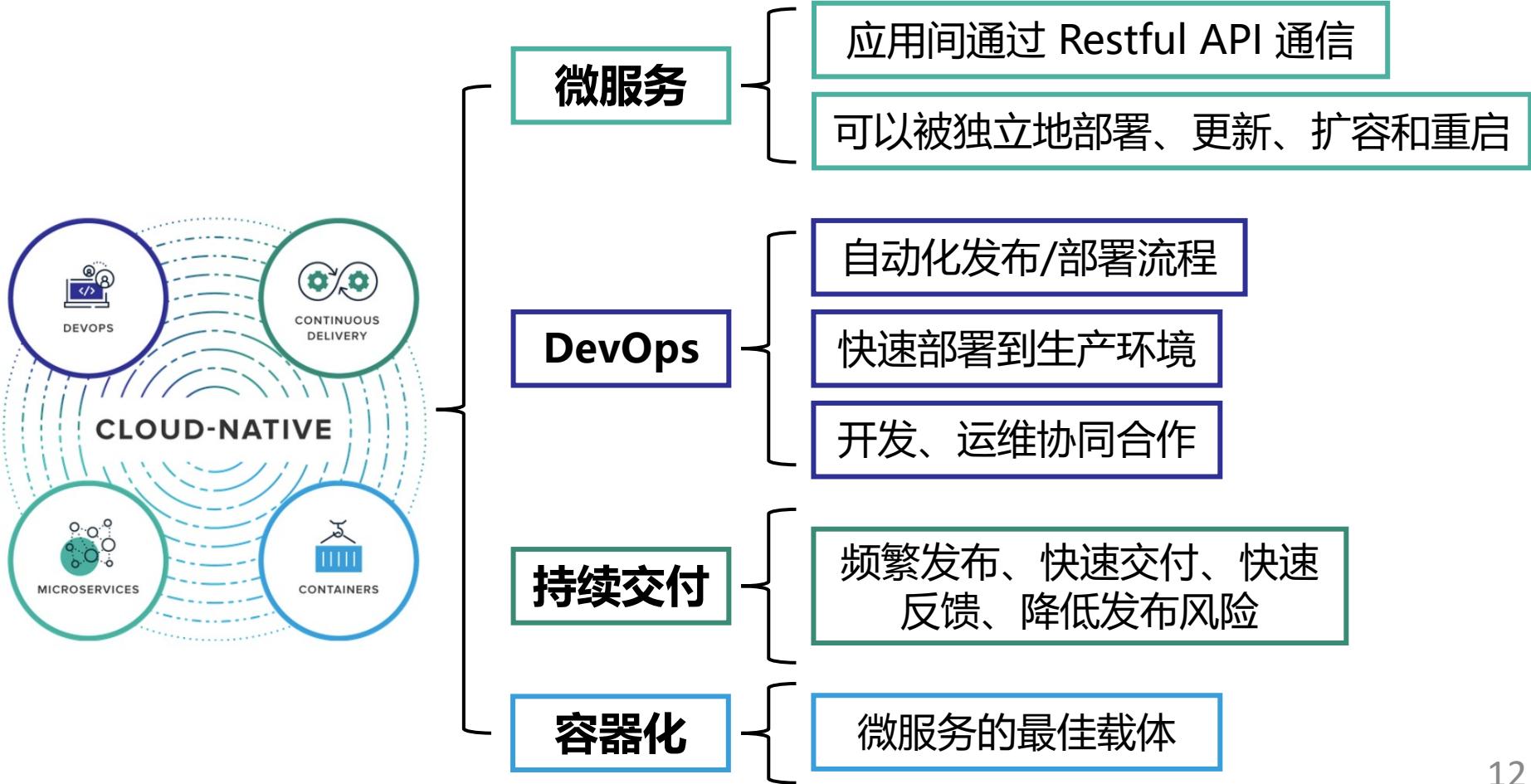
使用方法不同

云计算通常作为一种**服务模型**，用户可以按需购买和使用资源；云原生是一种**应用开发和运行的方法**，需要开发者具备一定技术能力

# 云原生代表技术

# 云原生代表技术

- 从宏观概念上讲，云原生是不同思想的集合，包含各种技术
- 这些技术能构建**容错性好、易于管理和便于观察**的松耦合系统



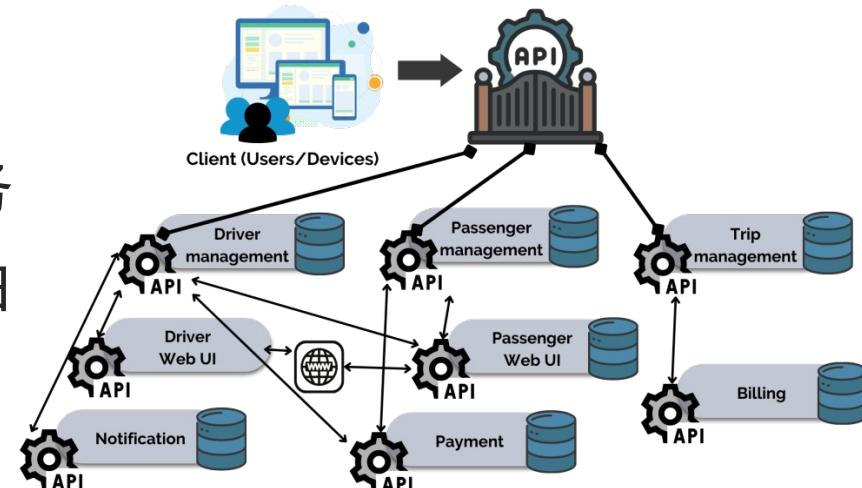
# 微服务

随着企业的业务发展，传统业务架构面临着很多问题

- 单体架构在需求增多时无法满足变更要求
- 系统经常会因为某处业务的瓶颈导致整个业务瘫痪
- 整体组织效率低下，无法很好地利用资源

微服务架构

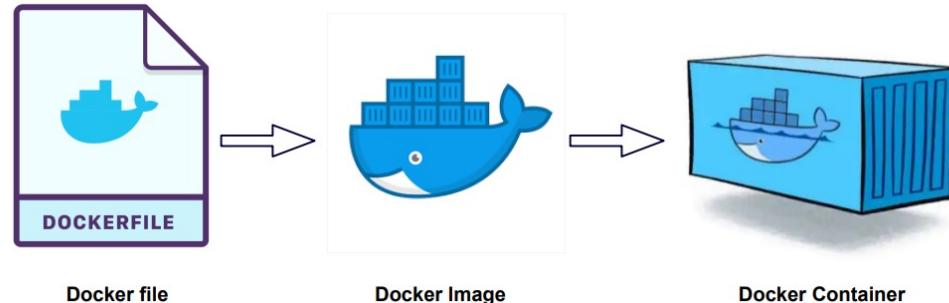
- 是一种架构风格，也是一种服务
- 颗粒较小，一个大型复杂应用由多个微服务组成
- 采用 Unix 设计的哲学
  - 每种服务只做一件事
  - 服务间松耦合的，能被独立开发、部署和升级



# 容器技术

- 一种轻量级的、可移植的、自包含的软件打包技术
- 容器技术的核心是提供一个隔离的运行环境

- 文件系统
- 网络配置
- 进程空间



Docker 已经成为容器技术的事实标准

## □ 特点

- 轻量级 (Lightweight): 容器共享宿主操作系统的内核
- 可移植性 (Portability): 容器内包含了应用及其所有的依赖项
- 隔离性 (Isolation): 每个容器都在自己的命名空间中运行
- 可扩展性 (Scalability): 容器可以很容易地进行水平扩展

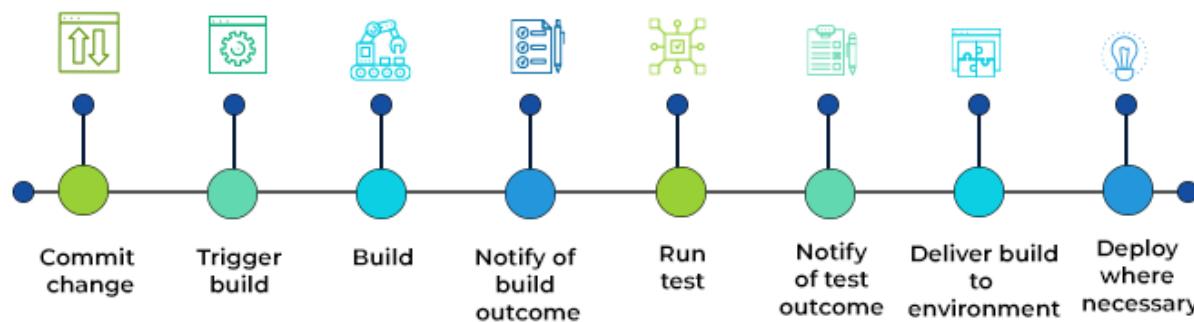
# 持续集成与持续交付技术

## 口持续集成 (Continuous Integration, CI)

- 开发团队将代码频繁地集成到主分支中
- 自动化的构建和测试，以尽早地发现和修复问题，提高软件质量

## 口持续交付 (Continuous Delivery, CD)

- 以可持续、自动化的方式将变更直接部署到生产环境
- 变更包括新功能开发、配置更改、Bug 修复等

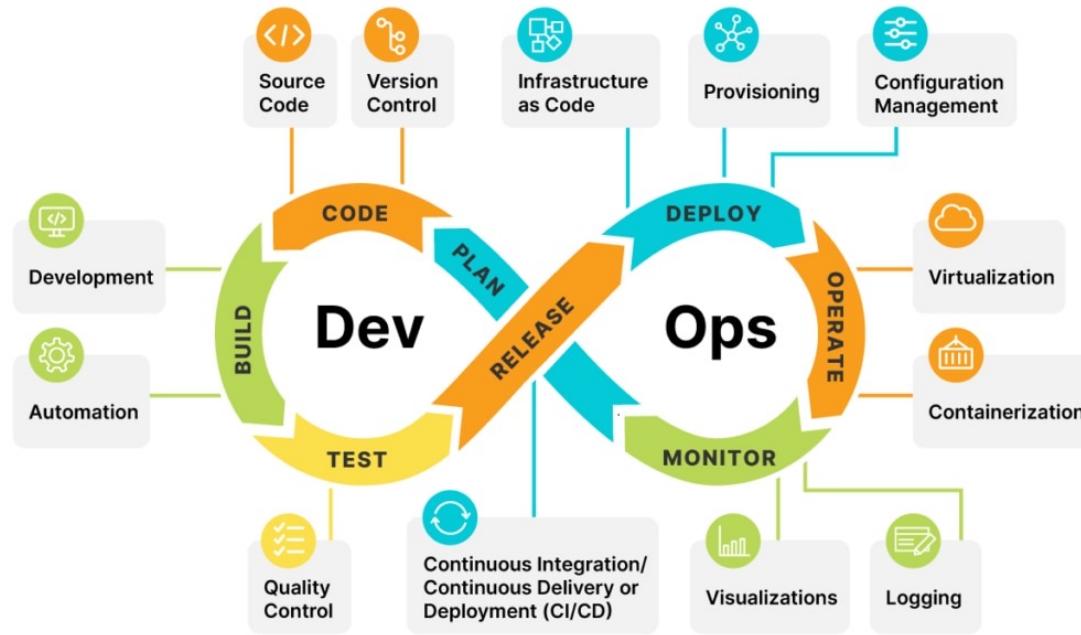


**CI/CD Pipeline**

# DevOps

DevOps 是一种软件开发方法，强调开发 (Development) 和运维 (Operations) 团队之间的高效合作

目的是打破传统的 “研发-部署-维护” 的隔离模式，加快软件的开发和交付过程，从而提高软件的质量和性能



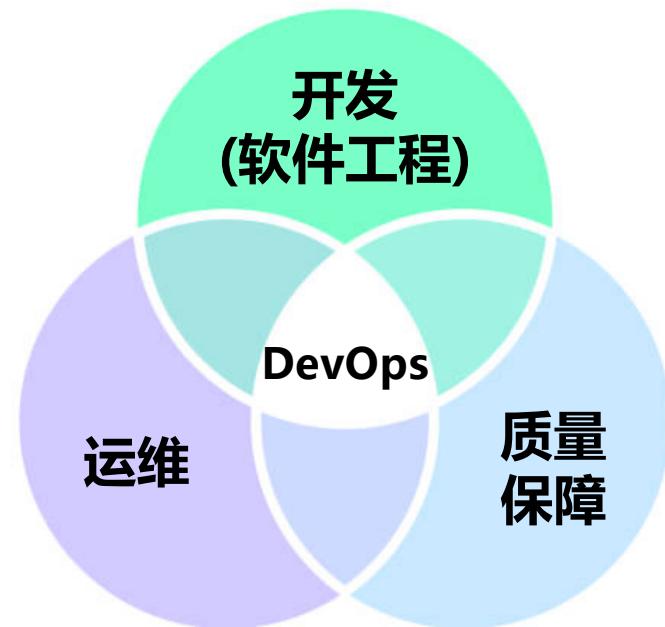
# DevOps

口字面上只是 Dev+Ops，实际是一组**过程、方法与系统**的统称

- 通过自动化软件交付和架构变更的流程 (CI/CD)，使得构建、测试、发布软件能够更加地快捷、频繁和可靠
- 组织架构、企业文化与理念等，需要自上而下设计，用于促进**开发部门、运维部门和质量保障部门**之间的沟通与协作
- **开发 (软件工程)、系统运维和质量保障三者的交集**



“Glue People” are  
also important



谷歌前 CEO 埃里克·施密特

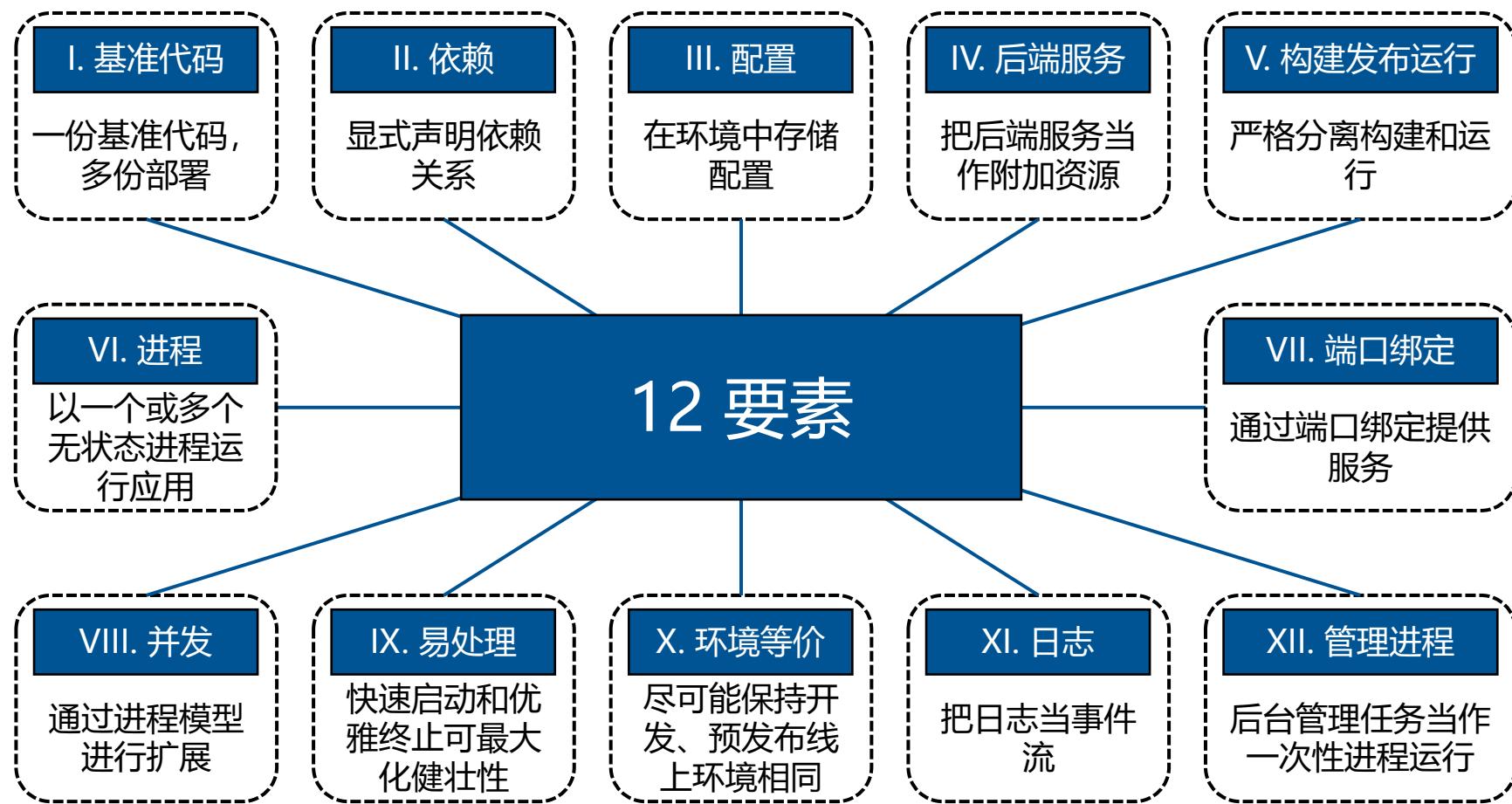
# 云原生归纳

- 口充分利用云计算技术的优势：采用**云端优先策略**，从云服务中获取最大价值
- 口实现**快速、敏捷、频繁的交付模式**
- 口通过技术创新更多地**扩展云计算技术的边界**
- 口云原生包含不同思想/技术，与云上应用架构应具备的特性几乎是——对应的：
  - DevOps、持续交付对应**更快的上线速度**，即**敏捷性**
  - 微服务对应**可扩展性及故障可恢复性**
  - 敏捷基础设施（容器化）对应**扩展能力的资源层支持**

# 云原生应用开发实践的 12 要素

# 云原生应用开发实践的 12 要素

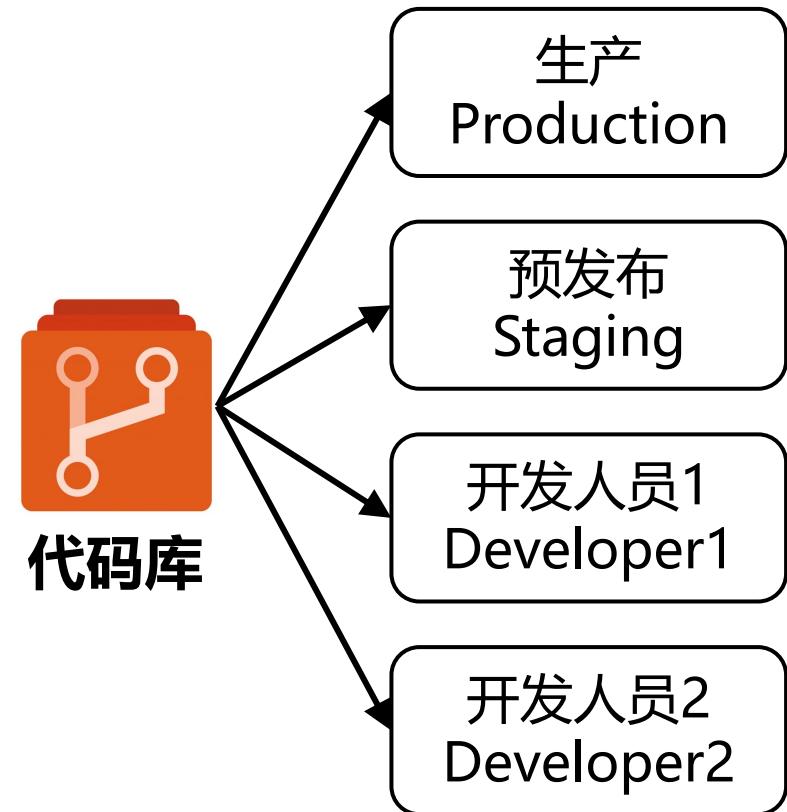
口云原生应用开发的 12 要素是由 Heroku 的联合创始人 Adam Wiggins 提出的，用于指导构建**可扩展和可维护**的云原生应用



# 原则 1：一份代码库与多份部署

应用只有一份基础代码库，并用版本控制系统（如 Git）追踪管理。所有的部署（包括生产环境、测试环境等）都应该是这份代码库的副本

- 口一旦有多个代码库，就不能称为一个应用，而是分布式系统
- 口多个应用共享一个代码库需要将共享的代码拆分为独立的类库
- 口尽管每个应用只对应一个代码库，但可以同时存在多份部署



# 原则 2：显式声明依赖关系

应用应该自行声明其直接依赖的所有库，  
以便创建明确、可重复的构建过程

- 是保持应用程序可移植性和一致性的主要步骤
- Dockerfiles 极大地改进了依赖声明，允许将所有内容打包到同一个镜像中，应用是自包含的 (**self-contained**)

```
FROM golang:1.13 as builder
ENV PATH /go/bin:/usr/local/go/bin:$PATH
ENV GOPATH /go
COPY . /go/src/github.com/liqotech/liqo
WORKDIR /go/src/github.com/liqotech/liqo
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go
build ./cmd/crd-replicator/
RUN cp crd-replicator /usr/bin/crd-replicator

FROM scratch
COPY --from=builder /usr/bin/crd-replicator /usr/bin/crd-
replicator
ENTRYPOINT [ "/usr/bin/crd-replicator"]
```

# 原则 3：在环境中存储配置

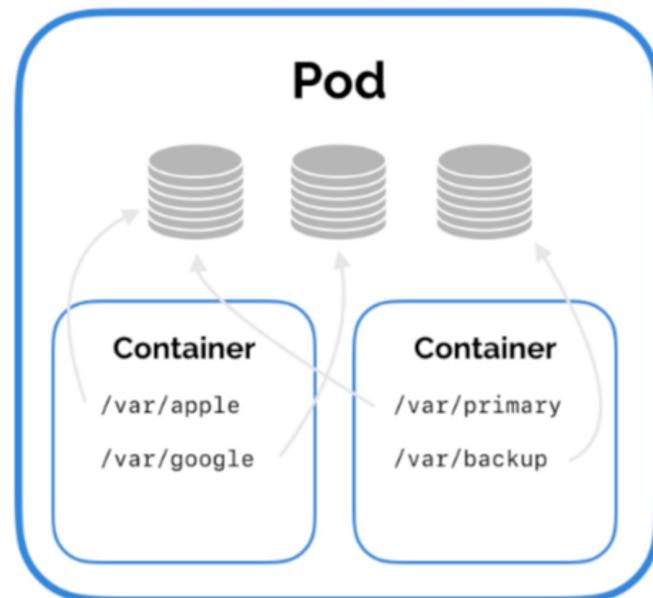
**应用的配置应该和代码分离，存储在环境中，而不是硬编码在代码里**

## 常见配置

- 资源句柄（如数据库、缓存）
- 凭据（密钥、密码）
- 服务的主机名和端口

## 优点

- 增加应用的可移植性
- 避免了敏感信息被误提交到代码库中
- 使得多份部署（开发、测试、生产等）更加简单



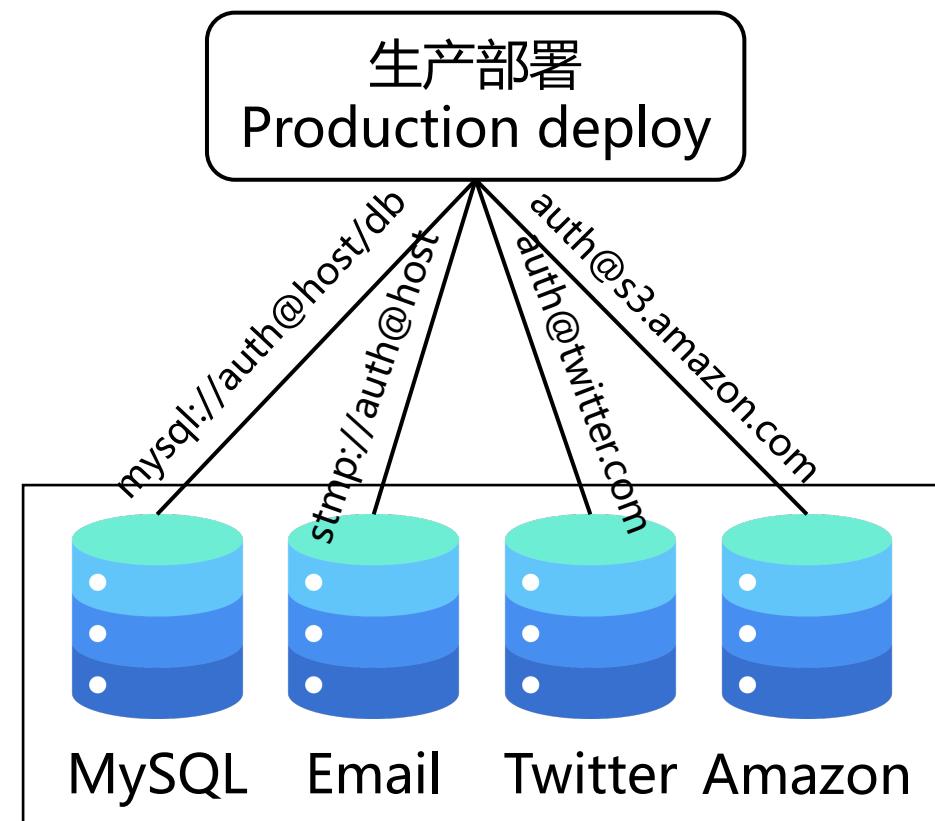
# 原则 4：把后端服务当作附加资源

应用应将所有的后端服务看作是附加的资源，这些资源可以通过网络，本地或者第三方服务进行访问，而且代码中并不假设这些后端服务存在的位置

## 口后端服务

- 如数据库 (Mysql CouchDB)
- 消息/队列系统 (RabbitMQ)
- SMTP 邮件发送服务 (Postix)
- 以及缓存系统 (Memcached)

## 口提高应用灵活性和可移植性的重要策略



# 原则 5：严格分离构建和运行

**应用的生命周期中的构建阶段和运行阶段  
应该严格分离**

□ 构建阶段

- 代码会被转换为可执行文件

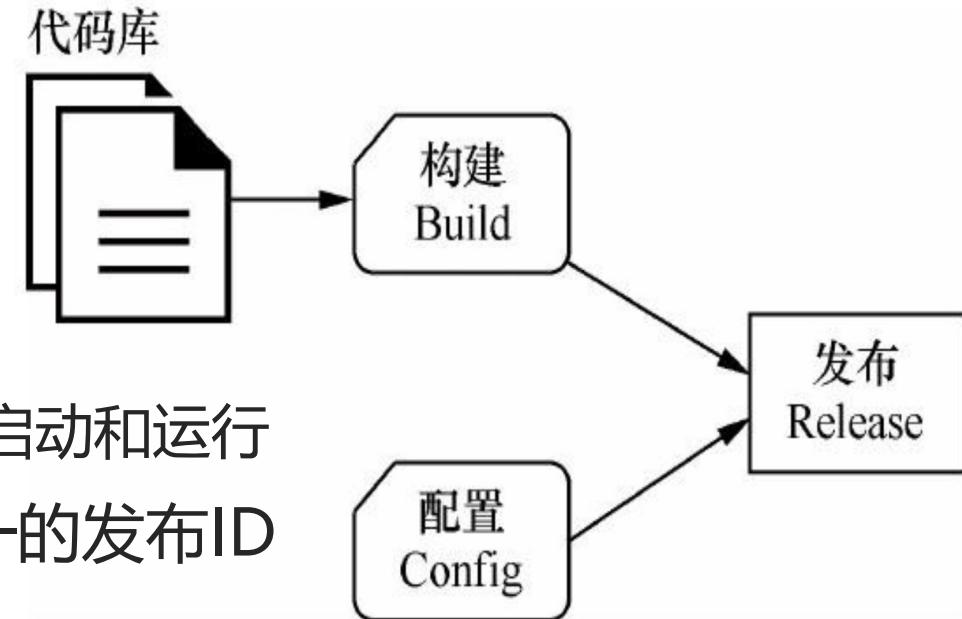
□ 运行阶段

- 可执行文件在适当的环境中启动和运行

□ 每一个发布版本对应一个唯一的发布ID

- 发布时的时间戳
- 增长的版本号码

□ 提高了部署的**可预测性、可重复性、安全性、效率**

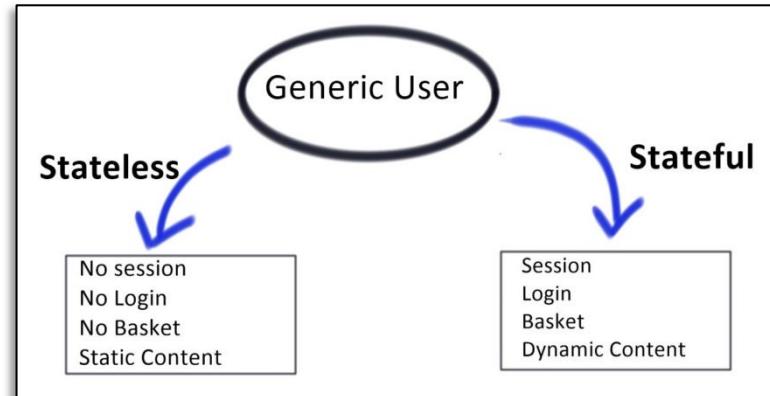


# 原则6：以一个或多个无状态进程运行应用

**应用的数据不应该在本地进程的内存中保存，而应该保存在外部的某个地方，如数据库或者缓存服务器中**

- 运行环境中，应用程序通常是以一个或多个进程运行的
- 进程应该是无状态的，不共享任何信息（比如 user sessions）
- 优点

- **可扩展性：**很容易的进行水平扩展，只需增加进程的数量即可
- **容错性：**如果一个无状态的进程失败了，只需启动新的进程
- **简单性：**无状态的进程比有状态的进程更容易理解和管理



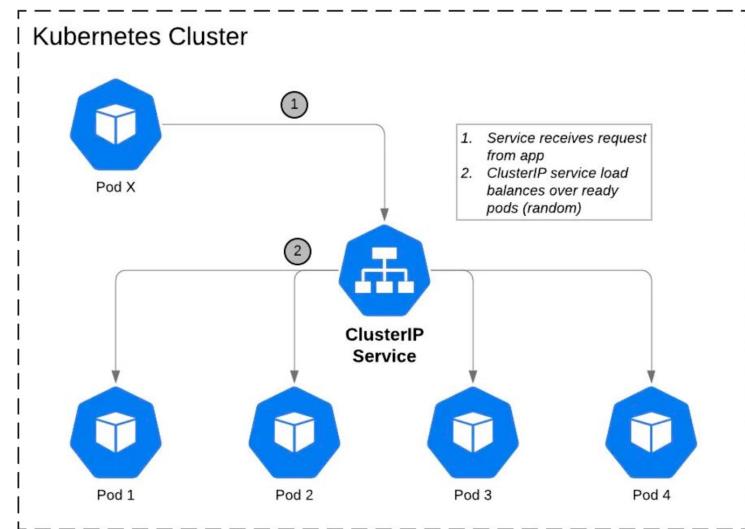
# 原则 7：通过端口绑定提供服务

应用不依赖于任何特定的网络服务器（如 Tomcat）来提供服务，而是直接与外部环境交互，自己成为一个服务

□ 就像一个店面直接开在街上，顾客可以直接进入，而不是开在大商场里，需要通过商场入口才能进入

□ 应用程序就是那个店面，而“街道”就是计算机的网络端口

□ 应用程序应该能够自己“开门做生意”，也就是自己监听某个网络端口，等待客户的请求



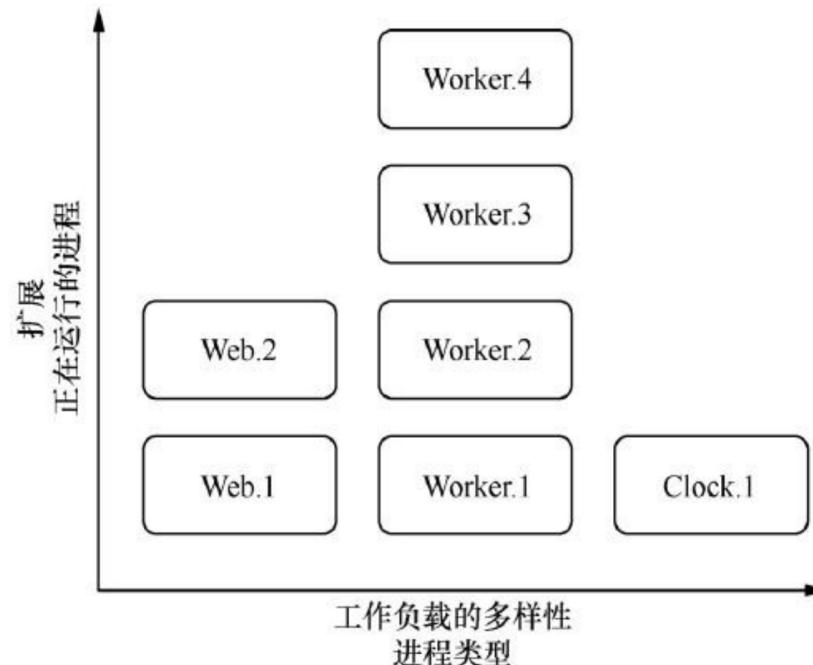
Kubernetes 服务完全符合此模式

# 原则 8：通过进程模型进行扩展

## 应用应该以扩展进程的方式来扩展性能和容量

口进程主要借鉴于 UNIX 守护进程模型。开发人员可以运用这个模型去设计应用架构，将不同的工作分配给不同类型的进程

口进程所具备的无共享、水平分区的特性意味着添加并发应用会变得简单而稳妥



# 原则9：快速启动和优雅终止可最大化健壮性

**应用程序应该能够尽快启动并且在接收到停止信号时能够优雅地终止**

## □ 快速启动

- 当启动一个新的进程来处理更多的负载，或者在部署新版本的应用程序时，应用程序应该能够尽快启动并开始提供服务

## □ 优雅终止

- 当应用程序接收到停止信号时，它应该先完成当前的任务（如处理当前的请求，完成当前的事务），然后再停止
- 防止数据丢失或者状态不一致，也可以提供更好的用户体验

# 原则 10：尽可能保持环境相同

**应用程序在所有环境中的行为应尽可能一致，包括开发环境、测试环境、生产环境等，以减少因环境差异引起的问题，提高开发和部署的效率**

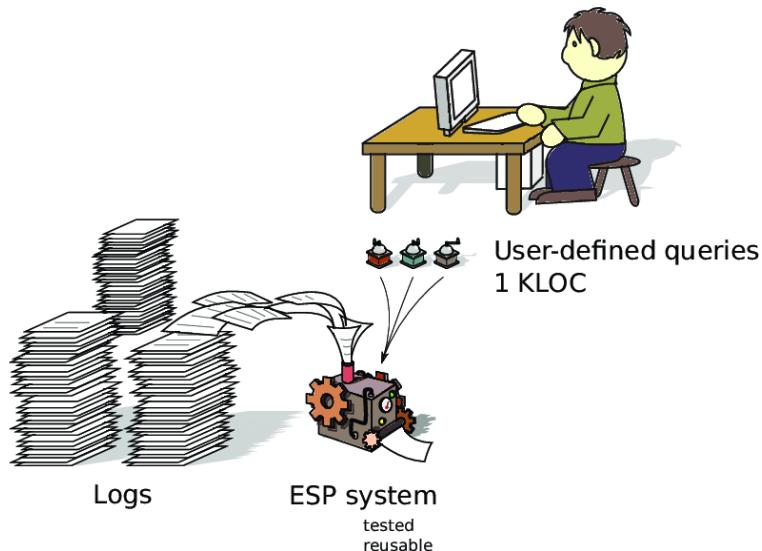
□ 开发和线上环境之间存在很多差异，表现在：

- **时间差异**：开发人员正在编写的代码可能需要几天、几周，甚至几个月才会上线
- **人员差异**：开发人员编写代码，运维人员部署代码。
- **工具差异**：开发人员使用 Nginx、SQLite、OS X，而线上环境使用 Apache、MySQL/ZLinux

# 原则 11：把日志当作事件流

将日志看作是一个或多个事件流，而不是管理文件。应用程序应该将事件写入到 `stdout`（标准输出），让运行应用程序的环境来决定如何处理这些输出

- 在现代云环境和容器环境中，基于文件的日志管理非常复杂
- 日志流简单、灵活，可以很好地适应不同的运行环境



# 原则 12：后台管理任务当作一次性进程运行

**应用程序中的后台管理任务应该被当作一次性进程来运行，  
包括数据库迁移、处理数据、发送邮件等**

□ 一次性进程意味着

- 这些进程在启动后执行一次任务，然后终止
- 这些进程应该是无状态的
- 简化应用程序的架构，提高应用程序的可扩展性和可维护性

# 云原生的落地：Kubernetes

# 云原生的落地：Kubernetes

## □ Kubernetes 是云原生哲学的体现

- 通过**容器技术和抽象的 IaaS 接口**，屏蔽了底层基础设施的细节和差异，便于企业**充分利用云的特性和优势**
- 是 Google 基于内部的 Borg 改造的一个**通用容器编排调度器**，于 2014 年开源，并于 2015 年捐赠给 Linux 基金会下属的云原生计算基金会 (CNCF)

## □ 容器编排调度 (Container Orchestration)

- 在容器化的环境中，如何管理和控制容器的生命周期，包括容器的部署、扩展、负载均衡、网络配置、安全性、故障恢复等
- 其他容器编排调度工具
  - Docker Swarm
  - Apache Mesos
  - Amazon ECS

# Kubernetes

开源的容器编排 (orchestration) 系统，用于自动化部署、扩展和管理容器化应用程序

口K8s 的主要功能包括：

- 服务发现和负载均衡
- 存储管理
- 自动部署和回滚
- 容器健康管理
- 密钥和配置管理



kubernetes

How to pronounce Kubernetes?

coo-ber-net-ees

(even when it is shortened as K8s)

# 容器回顾

## 容器

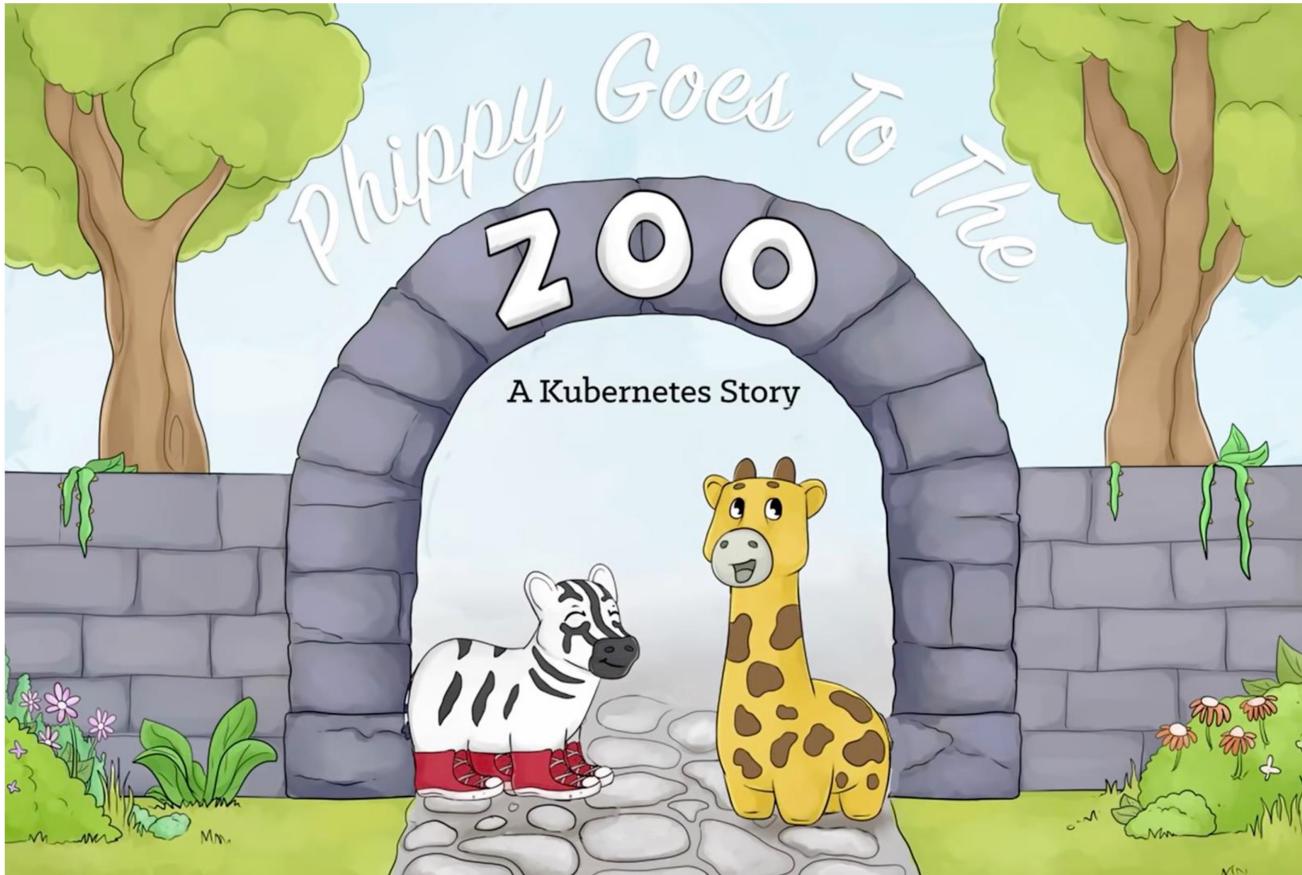


## 传统虚拟机



# Kubernetes 基础

口介绍动画 (Phippy Goes To The Zoo)



<https://www.bilibili.com/video/BV1su4y117c3>

<https://www.bilibili.com/video/BV1Du4m137pK>

<https://www.bilibili.com/video/BV1aA4m1w7Ew>

# Kubernetes 关键理念

## □ API 与实现解耦

- 只要符合 API 规范，那么相同对象就可以根据上下文（例如，云提供商）以不同的方式实现

## □ 声明式配置 (Declarative configuration)

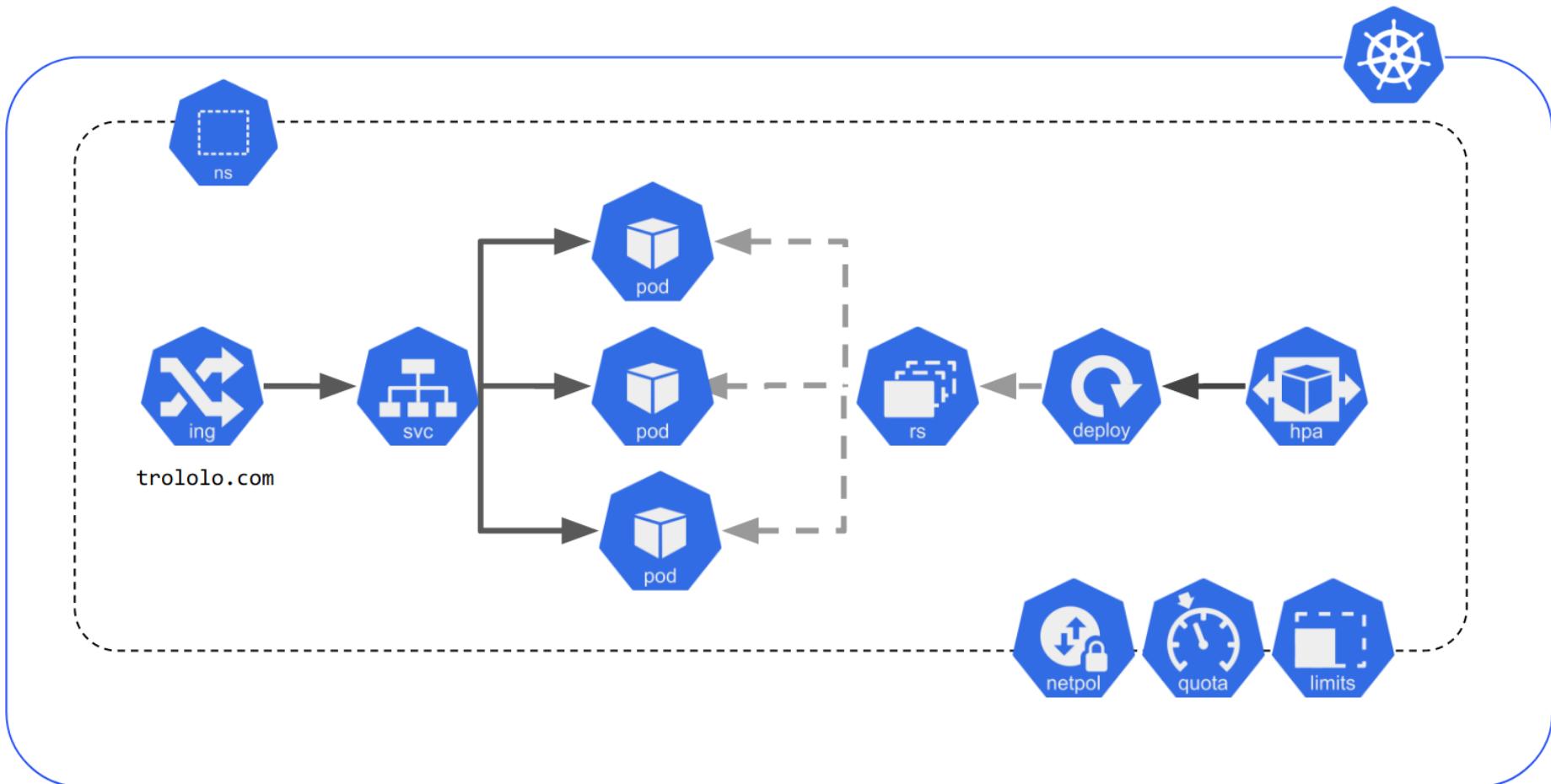
- 遵循“基础设施即代码 (Infrastructure as Code)”的理念
- 仅描述期望的系统状态，无需定义控制流

## □ 以控制循环为导向的方法 (Control Loop-oriented)

- 使对象的状态收敛于期望的状态
- 持续检查状态是否符合期望

# Basic Kubernetes Objects

## 口基本 Kubernetes 对象



# Kubernetes 对象

□ K8s 用来表示集群状态的持久化条目

□ 对象的类型由以下字段明确定义：

- apiVersion
- kind (Pod, Service, Deployment)

□ 每个对象都具有：

- **spec**: 对象的预期状态描述，通常由用户提供
- **status**: 当前状态，通常由系统更新

□ 对象有两种元数据：

- labels: 用户有关的信息，但不对核心系统具有语义含义
- annotations: 可以用来丰富基本规范的信息

```
apiVersion: VERSION # eg. v1
kind: TYPE # eg. Pod
metadata:
  name: NAME # eg. nginx
  annotations:
    # eg. k8s.v1.cni.cncf.io/networks: multus1
    test: true
  labels:
    color: blue # eg. release: stable
spec:
  # Expected status of the resource
status:
  # Current status of the resource
```

# Pod 基本介绍

□ Kubernetes 中**最小的资源部署和调度单位**

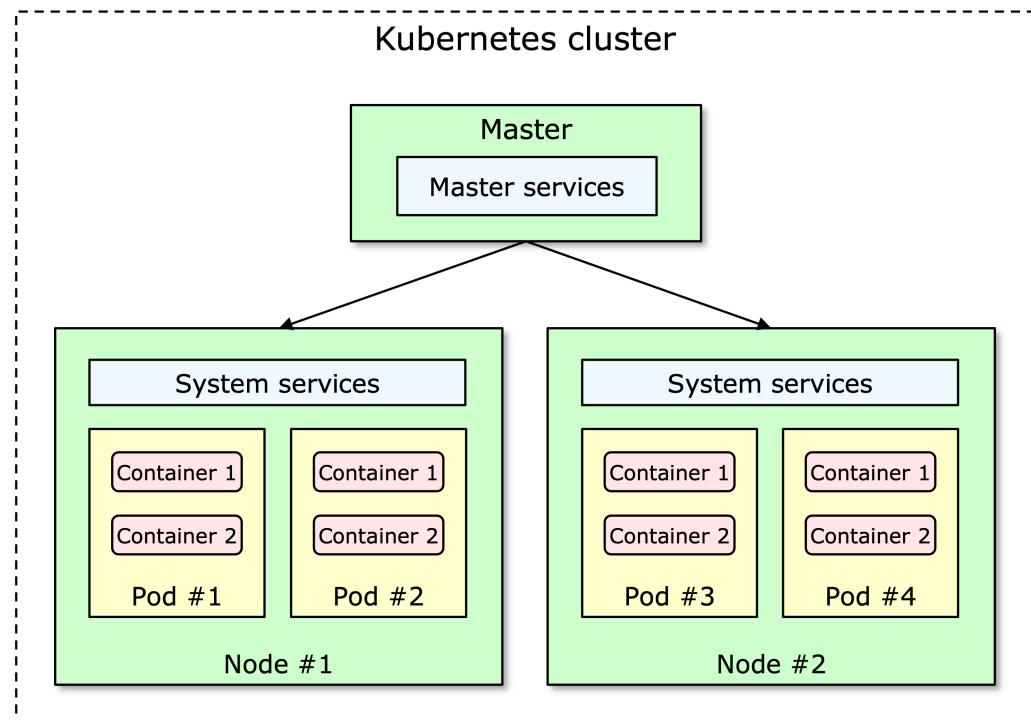
□ 包含一组高度耦合的容器和数据卷

□ 共享命名空间 (高效通信和资源共享)

- 共享 IP 地址和本地主机 (可以通过 “localhost” 进行通信)
- 共享 IPC 等

□ 被管理的生命周期

- 与节点绑定
- 可以死亡，但不能以相同的 ID 重启



# Pod

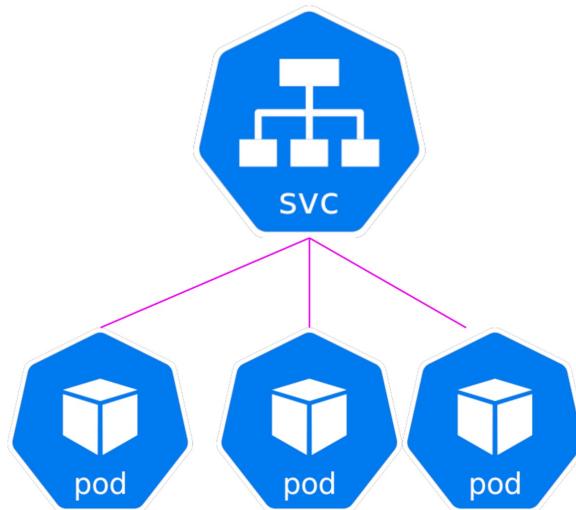
- Pod 是 Kubernetes 应用程序的基本执行单元
- 一个 Pod 可能由一个或多个容器组成，大多数 pod 是单容器的
- 一个 Pod 必须在单个节点上执行
- 在 Kubernetes 中，Pod 就像操作系统中的进程一样

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: nginx
  name: nginx-5f78746595-7zs8g
  namespace: default
spec:
  containers:
    - image: nginx
      name: nginx
      ports:
        - containerPort: 80
          name: http
          protocol: TCP
      volumeMounts:
        - mountPath: /var/log/
      volumes:
        - emptyDir: {}
          name: logs
```

# Service 服务

口在 K8s 中， Service 是用来**暴露应用服务的抽象概念**：

- Service 是一种抽象层，它定义了一组 Pod 的逻辑集合以及访问这些 Pod 的策略，实现负载均衡、服务发现等功能
- 当 Pod 的数量或位置发生变化时（阔缩容、故障），Service 会自动更新，提供一个**稳定的、可靠的和持续的网络接口**
- 可以把 Service 看作是 Pod 的一种接口，使得其他 Pod 或者外部用户可以与之通信，而**不需要关心其运行在哪些节点上**



# Service 服务类型

Service 类型	描述	访问方式	适用场景
ClusterIP	默认类型，在集群内部分配虚拟 IP，仅内部可访问	集群内通过服务名或 ClusterIP 访问	内部微服务间通信（如后端 API 调用）
NodePort	在每个节点上开放固定端口（范围：30000-32767），外部可通过 NodeIP:Port 访问	外部通过任一节点 IP 和指定端口访问	测试环境或小规模服务暴露（无需外部负载均衡器）
LoadBalancer	使用云提供商的负载均衡器（如 AWS ELB、GCP Load Balancer），自动分配外部 IP	外部通过负载均衡器 IP 访问	生产环境大规模服务暴露，需高可用性和流量管理
ExternalName	将服务映射到外部域名（如 db.example.com），不关联 Pod	集群内通过服务名访问，实际解析到外部 DNS	访问集群外服务（如第三方 API 或数据库），简化内部调用路径

# Volume 数据卷

- Pod 中能被多个容器访问的共享目录
- 定义在 Pod 之上，被一个 Pod 里的多个容器挂载到具体的文件目录之下；与 Pod 生命周期相同
- 可以让一个 Pod 里的多个容器共享文件、让容器的数据写到宿主机的磁盘上或者写文件到共享存储中

```
template:  
  metadata:  
    labels:  
      app: app-demo  
      tier: frontend  
  spec:  
    volumes:  
      - name: datavol  
        emptyDir: {}  
    containers:  
      - name: tomcat-demo  
        image: tomcat  
        volumeMounts:  
          - mountPath: /mydata  
            name: datavol  
        imagePullPolicy: IfNotPresent
```

# Namespace 命名空间

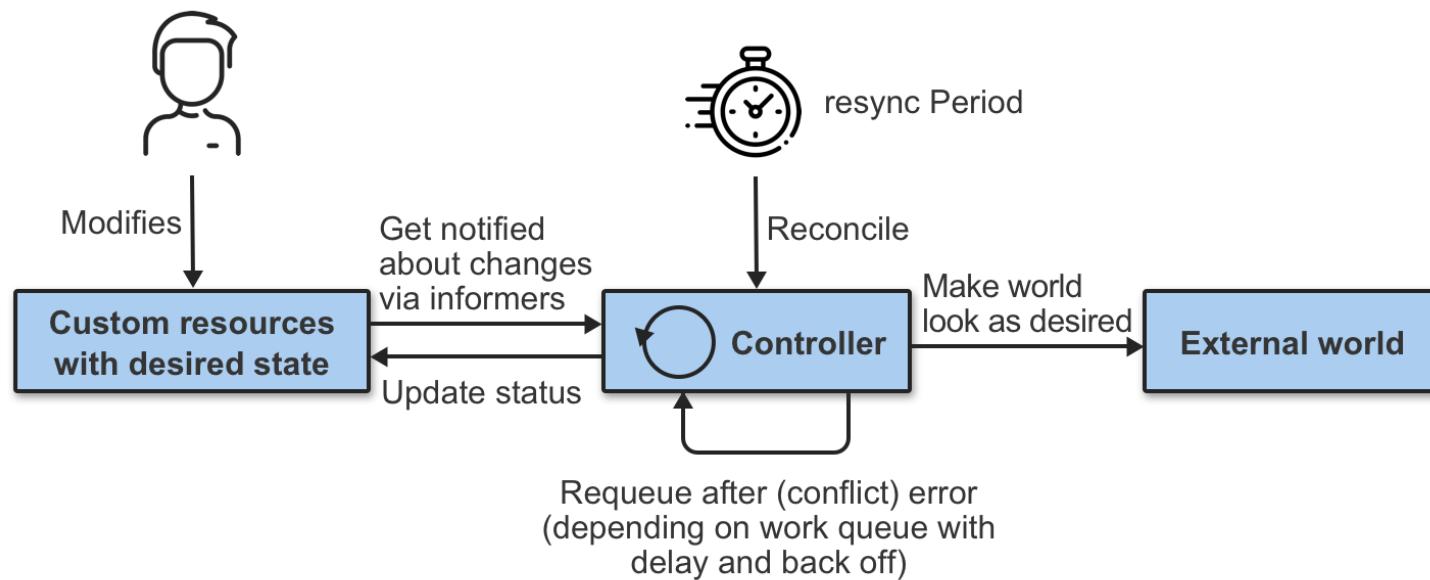
- 命名空间资源引入了"逻辑集群 (logical clusters)"的抽象概念
- 用于将物理集群划分为多个逻辑集群
- 每个 Namespace 都有自己的资源配置、角色和策略
- 多个用户、团队或项目可以在同一物理集群上独立工作

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

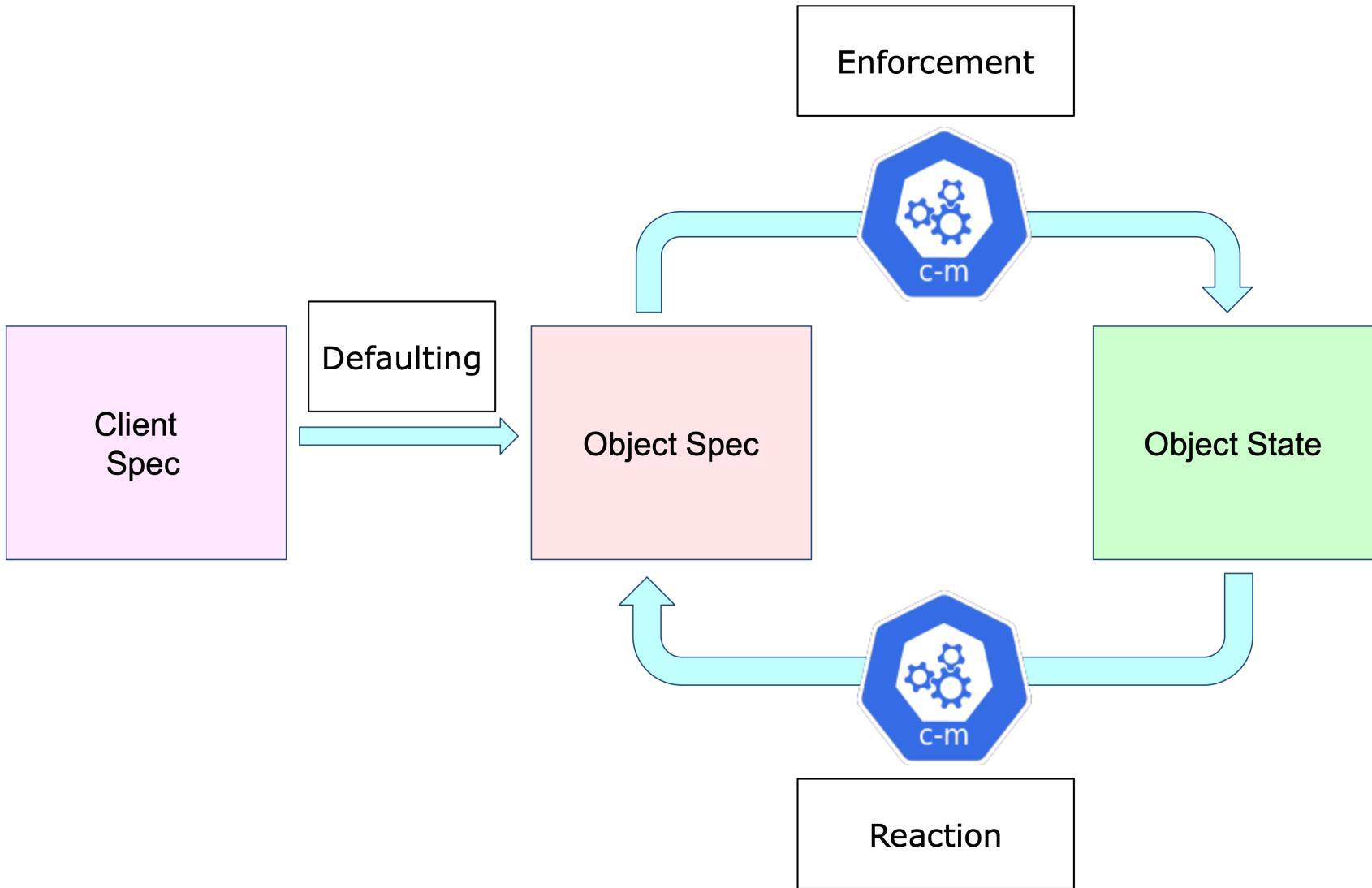


# Kubernetes Controllers 控制器

- 口在 K8s 中，控制器是一种核心概念，用于系统的自动化管理
- 口控制器是一种控制循环，它监视集群的某种状态，并确保该状态与用户定义的期望状态一致，如果当前状态与期望状态不一致，控制器将采取行动来修正
- 口除了内置资源（如 Pod），控制器还监视和操纵用户自定义资源



# 一种面向状态的方法



# "宠物 vs 牛群" 模型

口传统的基础设施模型将服务器或虚拟机视为**"宠物"**

口若服务器发生故障，我们会修复并尝试重新启动

口在这个模型中，服务器是**唯一的和不可或缺的**



口**"牛群"**模型取代了传统模型

口服务器只是一个副本，若发生故障，  
**将被销毁并创建一个新的副本**

口这是为故障而设计的：

- 副本的故障不应影响整个服务的功能

Unique servers, Virtual Machines



Replaceable servers and VMs

# ReplicaSet (rs) 控制器

口牛群模式在 Kubernetes 中通过 ReplicaSet 对象实现

口确保任何时间都有指定数量的 Pod 副本在运行；太多 Pod 会停止，数量不足则启动

口selector/labels 指定了需要监视的副本

- 若一个副本变得不可用（例如，节点故障）
- 新的副本将在其他地方生成

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - name: nginx
          image: nginx
```

# Deployment (deploy) 控制器

- 管理 Pod 和 ReplicaSet 的生命周期
- 描述期望的应用状态，包括应用的版本、副本数等，Deployment 会创建一个 ReplicaSet 来实现这个期望状态
- 尽管二者都能保证运行指定数量的 Pod，但 Deployment 提供了更高级的功能，如更新策略、版本回滚；ReplicaSet 则更基础，主要保证运行的 Pod 数量达到预期
- 在日常使用中，通常直接使用 Deployment，而不需要操作 ReplicaSet

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  spec:
    replicas: 5
    selector:
      matchLabels:
        app: toto
    template:
      metadata:
        labels:
          app: toto
      spec:
        containers:
        - name: toto
          image: titi/toto
          ports:
          - containerPort: 80
```

# Horizontal Pod Autoscaler (HPA)

□ Deployment 和 StatefulSets 默认有一个静态的副本数，可能不适合实际负载

□ Kubernetes 提出 HPA 对象对副本进行自动的扩缩容

□ HPA 定义了：

- 扩缩容的边界 (最大/最小的副本数)
- 扩缩容的对象
- 比较指标

□ 指标可以是

- 通过 kubelet/cAdvisor 接口获得的 CPU/memory 消耗
- 通过外部源 (如 Prometheus) 获取的应用程序指标

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

# DaemonSet 控制器

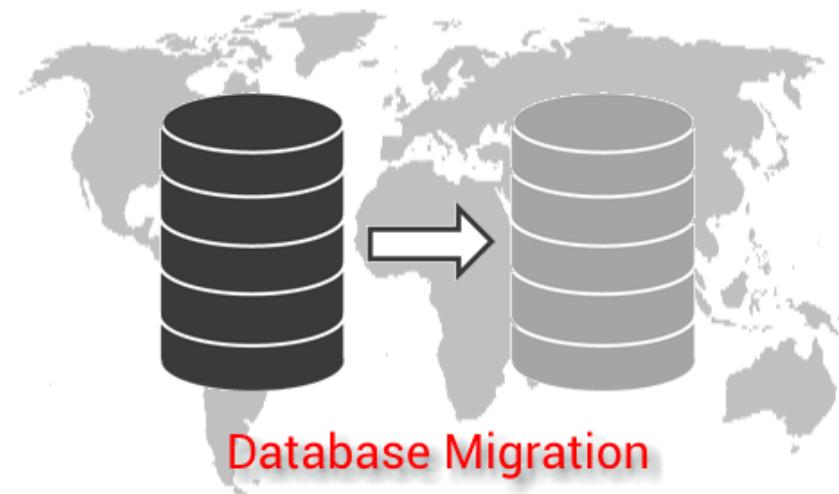
- 确保所有（或某些）节点上都运行一个 Pod 的副本
- 当集群中添加/删除节点时，该节点也会添加/回收 Pod；删除 DaemonSet 将会删除由它创建的所有 Pod
  
- DaemonSet 的一些常见用途：
  - **运行集群存储守护程序**：例如，在每个节点上运行 glusterd, ceph 等
  - **运行日志收集和监控守护程序**：例如，在每个节点上运行 fluentd, prometheus-node-exporter 等
  - **运行特殊用途的守护程序**：例如，在每个节点上运行用于硬件监控或配置的守护程序

# StatefulSets 控制器

□ 用于管理有状态应用，为 Pod 提供了唯一且持久的标识符，  
使其能够在节点重启或调度中保持其状态

□ StatefulSet 的一些主要特性和用途：

- 稳定、唯一的网络标识符
- 稳定、持久的存储
- 有序部署、扩展、删除、滚动更新



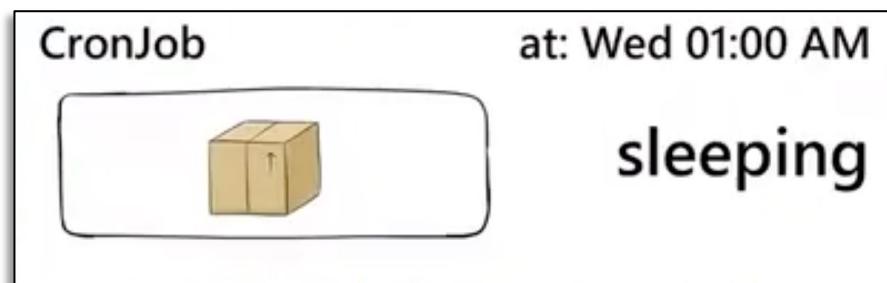
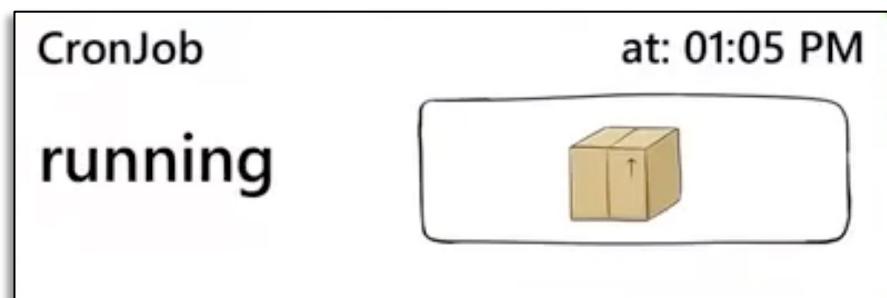
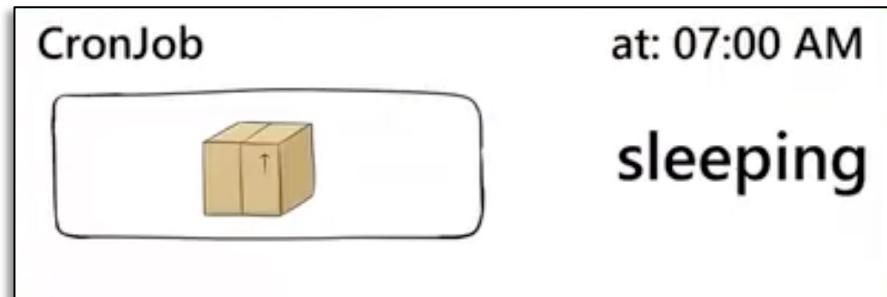
# CronJob 控制器

□ 在特定的预定时间或间隔执行作业

- 比如备份
- 发送邮件
- ...

□ CronJob 在特定的预定时间  
执行的是 Job

□ Job 本身负责管理 Pod 的  
生命周期



# ConfigMap/Secrets

□ ConfigMap 和 Secrets 是用于存储配置信息的两种资源对象

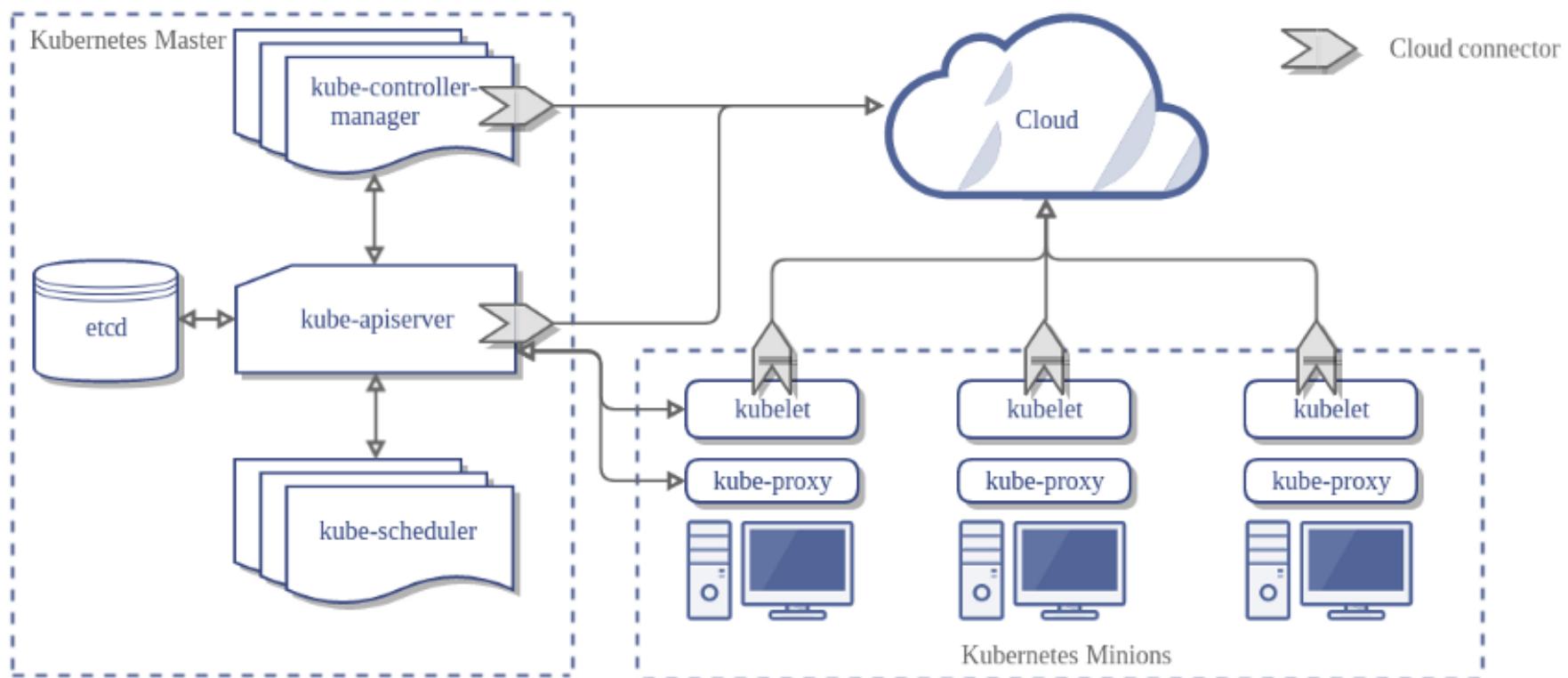
- ConfigMap 存储非敏感的配置信息，如配置文件、命令行参数
- Secrets 来存储敏感的配置信息，如密码、密钥、令牌

□ 均以键值对的形式存储 (Secrets 会加密)，可以在 Pod 中引用

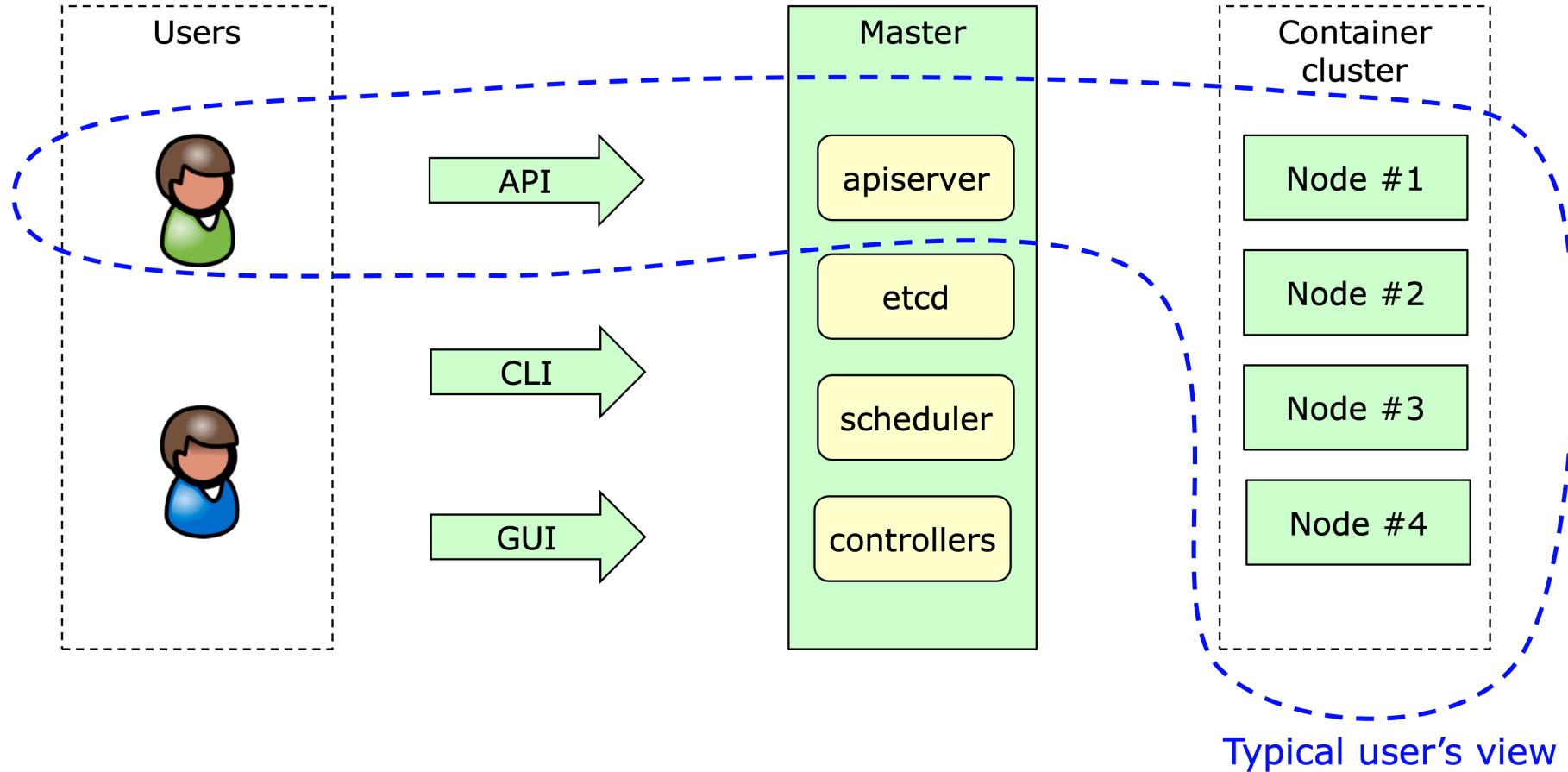
**将配置信息与 Pod 和其他资源分离，更好地管理和维护应用的配置，使得应用的部署和更新更加方便和灵活**



# Kubernetes 软件架构



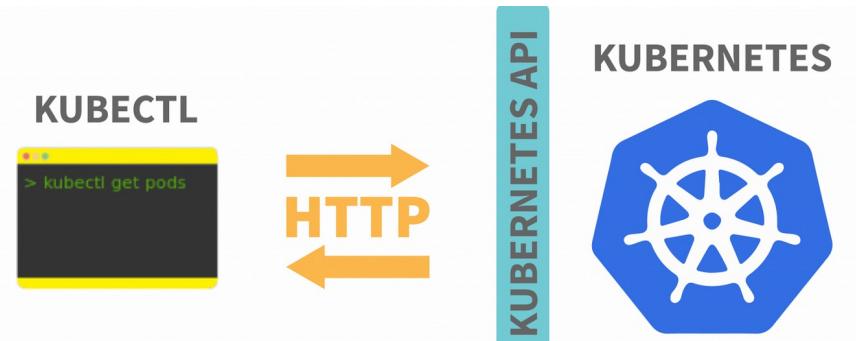
# Kubernetes 软件架构：用户视角



# Kubectl：K8s 的瑞士军刀

□ K8s 的命令行接口，是与 Kubernetes 集群交互的主要工具

- 通过 Kubectl，用户可以创建、查看、修改、删除各种资源对象，比如 Pod、Service、Deployment 等
- 其他高级功能：查看集群的状态、检查 Pod 的日志、执行集群中的 Pod 等



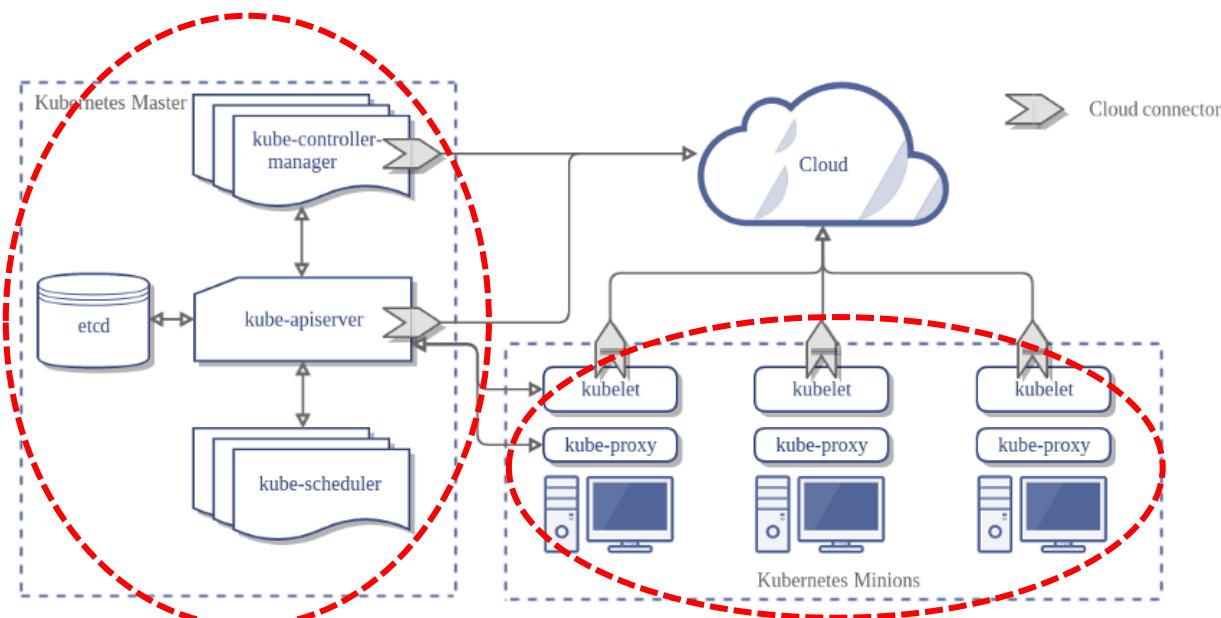
□ Kubectl 的使用方式非常灵活

- 在命令行中输入命令，如 “kubectl create” 创建一个新的 Pod
- 用 YAML 或 JSON 文件来描述资源对象，然后用 “kubectl apply” 命令将其应用到集群中

# 软件架构：概述

口在 K8s 中，主要有两类组件

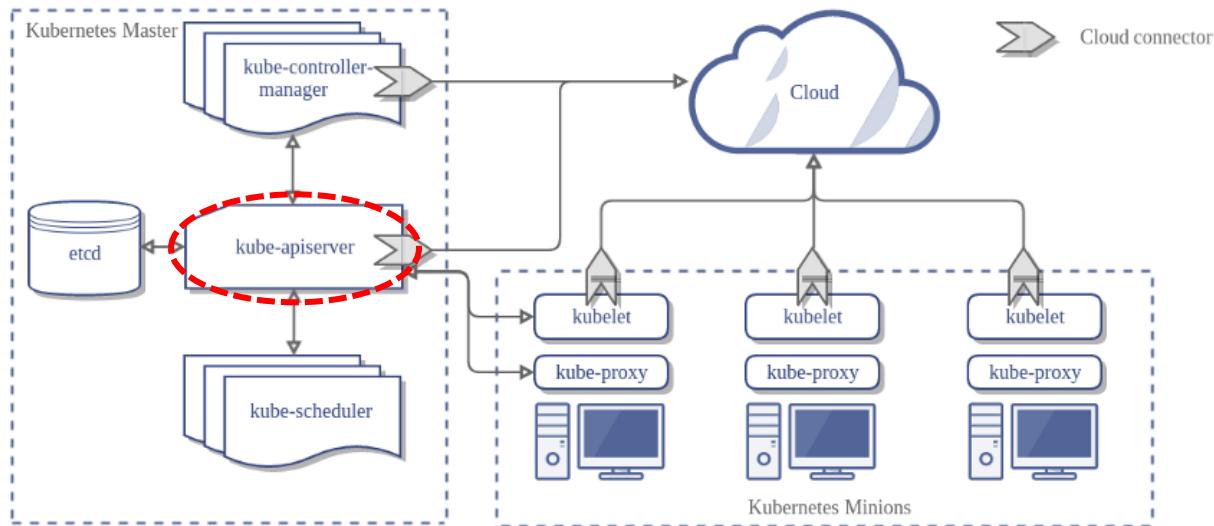
- **Master 组件**: 是 K8s 集群的控制中心，负责整个集群的管理和协调，包括 Kube-apiserver、etcd、Kube-scheduler、Kube-controller-manager
- **Node 组件**: 是 K8s 集群的工作节点，用于运行和管理 Pod，包括 Kubelet、Kube-proxy、容器运行时



# Kube-apiserver

□ Kube-apiserver 是 K8s 集群的前端，所有的系统组件和用户操作都通过它来进行交互

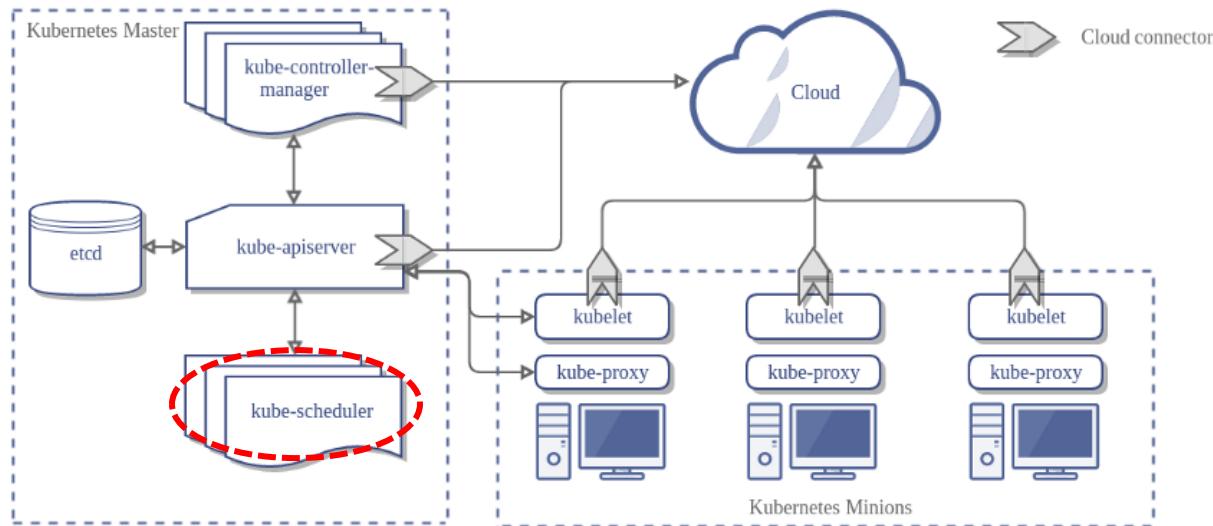
- 提供了 **HTTP REST 接口**，用于处理 API 请求并将其转发到适当的持久化层 (通常是 etcd)
- 同时也负责控制访问，包括身份验证、授权和访问控制



# kube-scheduler

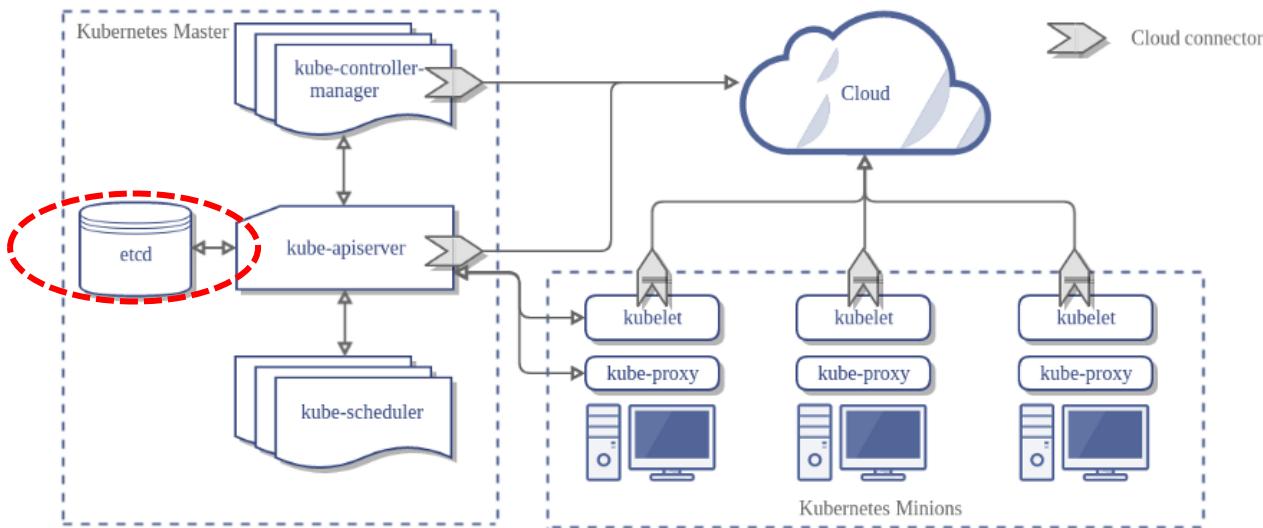
□ Kube-scheduler 是 K8s 的调度组件，负责决定新创建的 Pod 将在哪个 Node 上运行

- 根据各种考虑因素进行调度，比如资源需求 (CPU、内存等)、硬件/软件/策略约束、负载平衡、数据位置等
- Kube-scheduler 会持续监控集群状态，确保 Pod 的部署满足这些约束条件



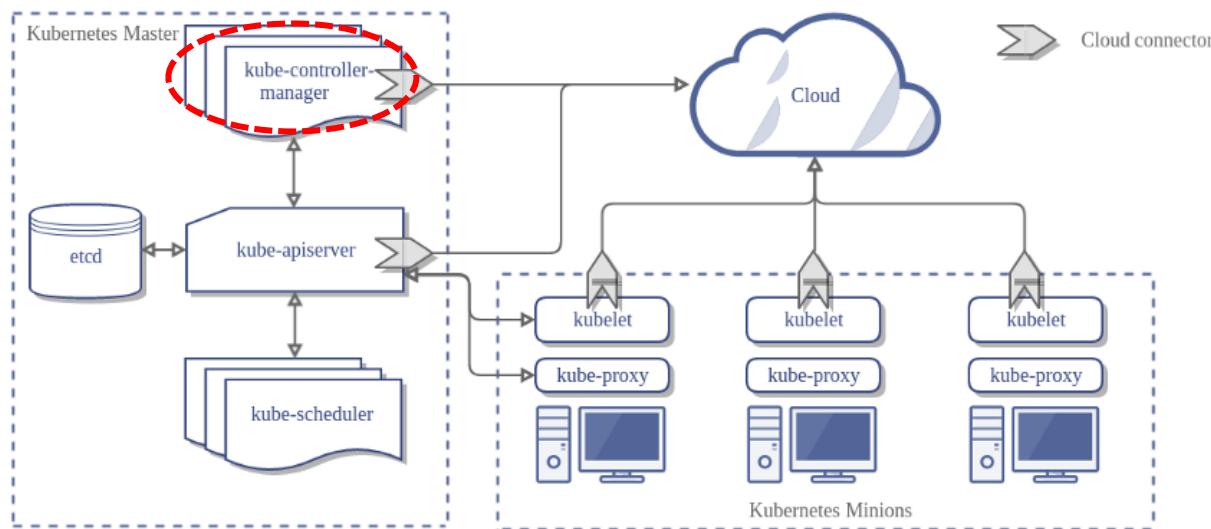
# etcd

- etcd 是一个**轻量级的、分布式的键值存储系统**，用于保存所有集群数据，包括节点、Pod、配置等的状态信息
- etcd 的数据可以通过 Kube-apiserver 访问，具有如下特点：
  - 快速 (每秒 10K 写入)
  - 安全 (带有 TLS)
  - 简单 (基于 REST 的 API)
  - 可靠 (使用 Raft 共识算法)



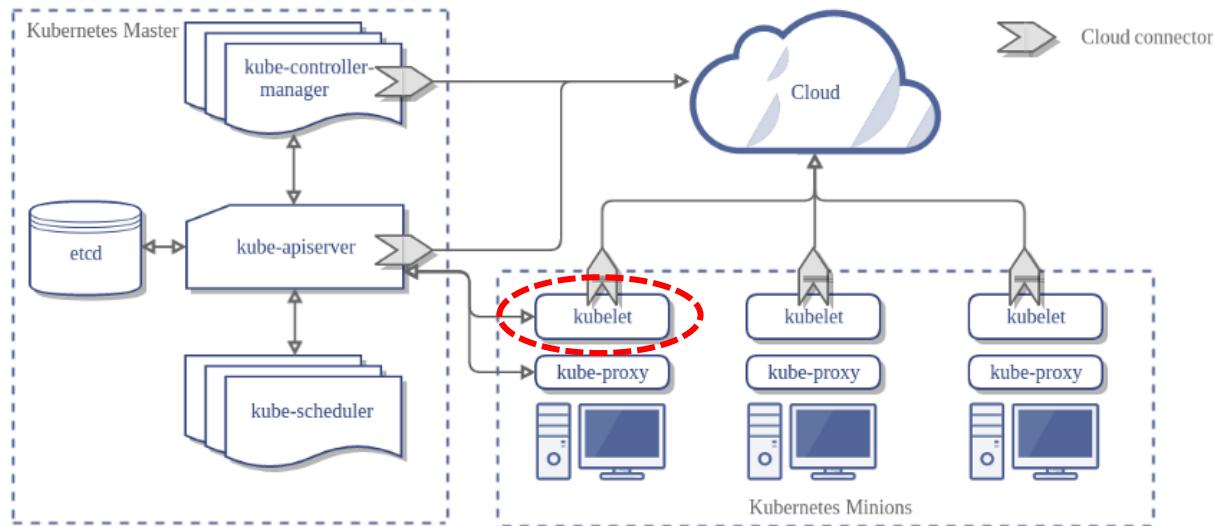
# kube-controller-manager

- Kube-controller-manager **运行一系列的控制器** (以进程方式运行), 这些控制器是 K8s 的核心控制循环
- 通过 kube-apiserver 监控整个集群的状态, 并确保当前的状态与预期的状态一致, 主要包括以下控制器:
  - 节点 (Node): 在节点出现故障时进行通知和响应
  - 副本 (Replication): 维护系统中运行的所有副本 Pod 的数量
  - ...



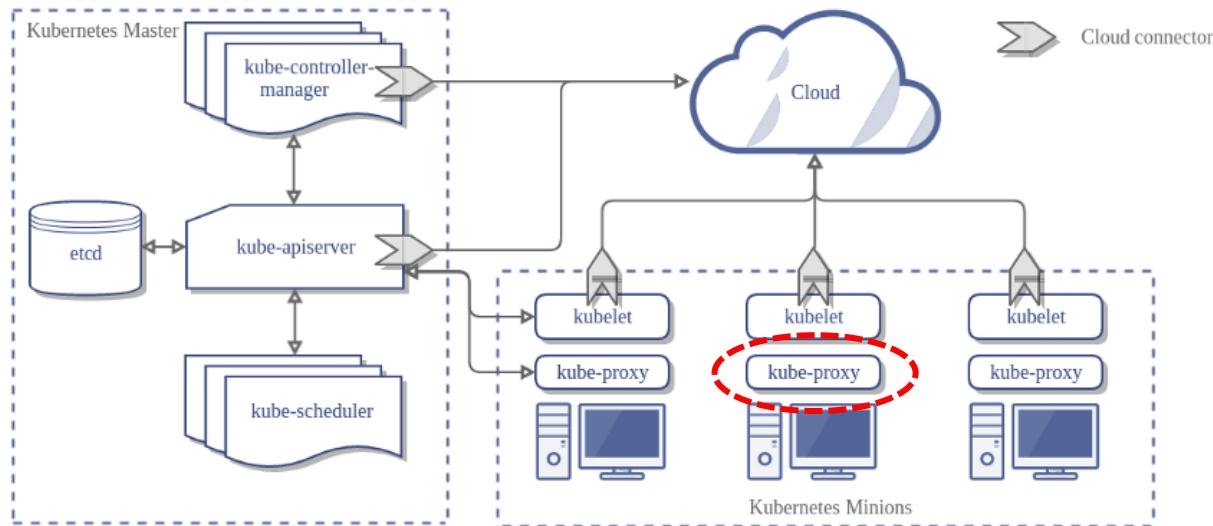
# kubelet

- 口Kubelet 是运行在每个工作节点上的**主要节点代理**, 负责保证容器在 pod 中正常运行
- 口收到来自 Kube-apiserver 的 PodSpecs (Pod 的描述和规格), 并确保 PodSpecs 中描述的容器已正确运行并且处于健康状态
- 口如果 Pod 中的容器因任何原因停止, Kubelet 将重新启动它们, 使其保持在预期的状态



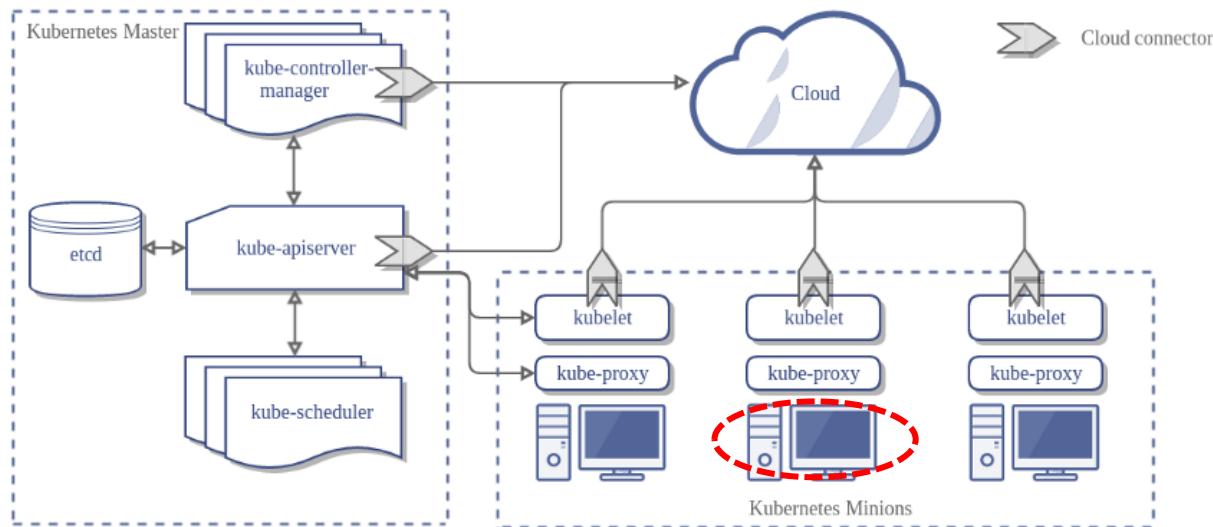
# kube-proxy

- Kube-proxy 运行在每个工作节点上，负责为 K8s 服务实现**网络代理功能**，包括 TCP、UDP 和 SCTP 的网络转发
- Kube-proxy 使用操作系统的包过滤层进行连接转发，也可以在用户空间进行转发
- Kube-proxy 负责实现 K8s 服务的 ServiceModel，使得 Pod 可以通过负载均衡访问服务



# Container runtime 容器运行时

- Container runtime 是**负责运行容器的软件**, K8s 支持多种容器运行时, 包括 Docker、containerd、CRI-O 等
- 容器运行时会根据 Kubelet 的指令, 启动或停止容器, **提供容器的运行环境, 并管理容器的生命周期和资源**





中山大學 软件工程学院  
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

谢谢

陈壮彬  
软件工程学院  
[chenzhb36@mail.sysu.edu.cn](mailto:chenzhb36@mail.sysu.edu.cn)