中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

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”

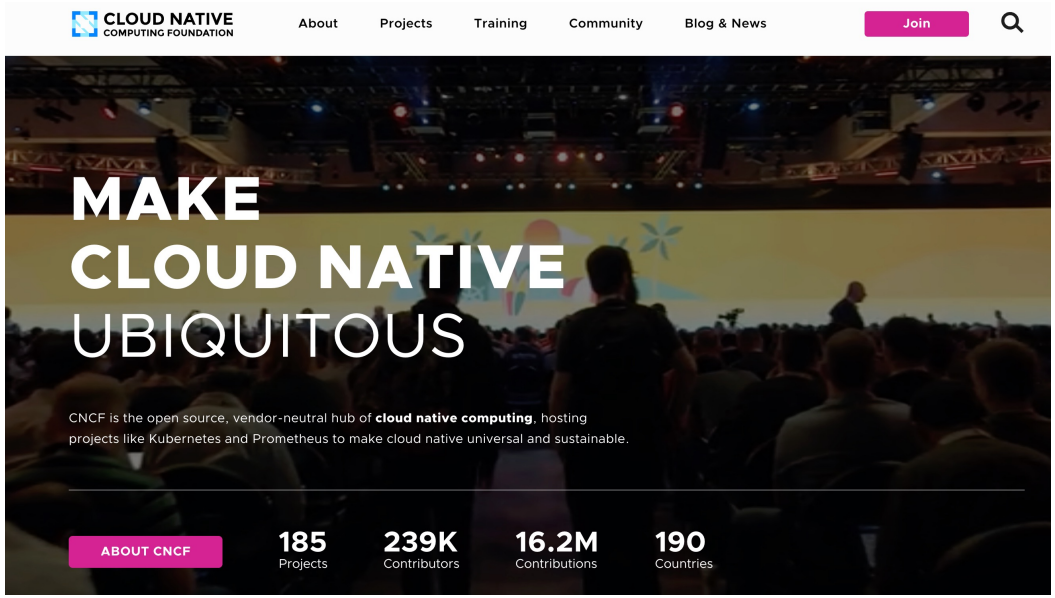
CNCF Cloud Native Definition v1.1

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

云原生计算基金会
2025年成立



https://www.cncf.io/
https://github.com/cncf/toc/blob/main/DEFINITION.md

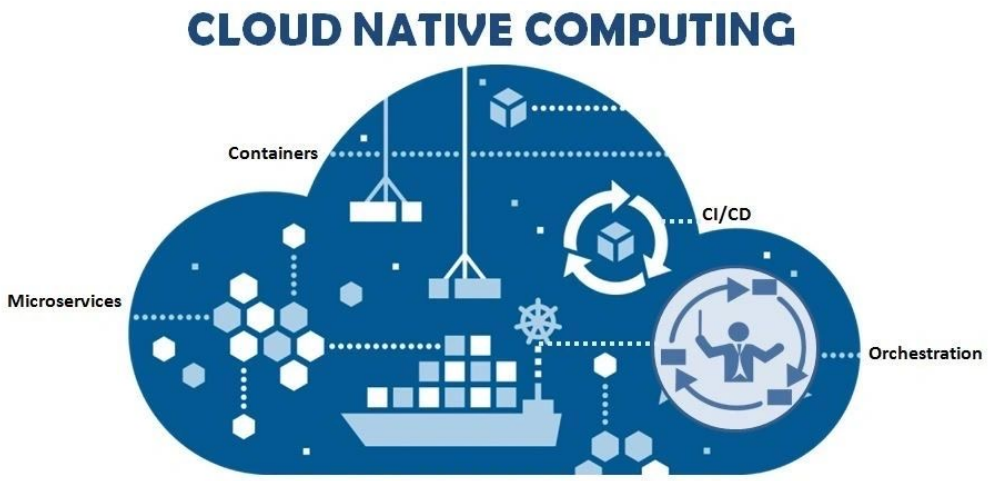
Native
本地的、原生的

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



- 弹性和可扩展性
- 快速迭代和创新
- 故障隔离与恢复
- 开发和运维效率
- 成本效益

云原生为什么会出现?

□互联网应用爆炸式增长，对软件交付提出了全新挑战

- 用户规模：从百万级到数十亿级（如微信、淘宝、TikTok)
- 迭代速度：从每年发布几次，到每天发布数十次甚至上百次
- 可靠性要求：7 × 24 小时不间断服务，分钟级故障恢复

□传统软件架构与运维方式难以满足需求

- 单体应用难以扩展，一次发布牵一发动全身
- 虚拟机启动慢、资源占用高，难以快速弹性扩缩容
- 开发和运维割裂，沟通成本高、发布周期长

□云计算的成熟为新一代架构提供了基础设施

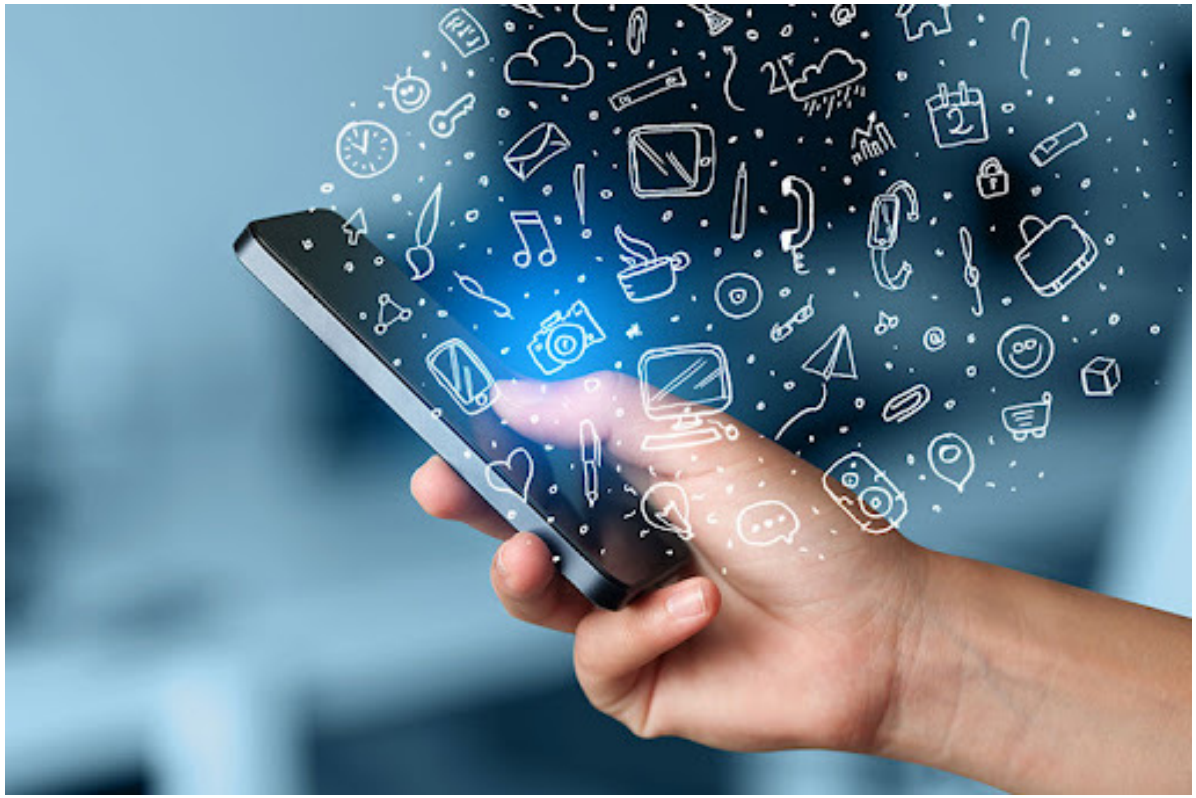
- 按需付费、弹性扩展、全球可用，让“为云设计”的应用成为可能

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
  - …

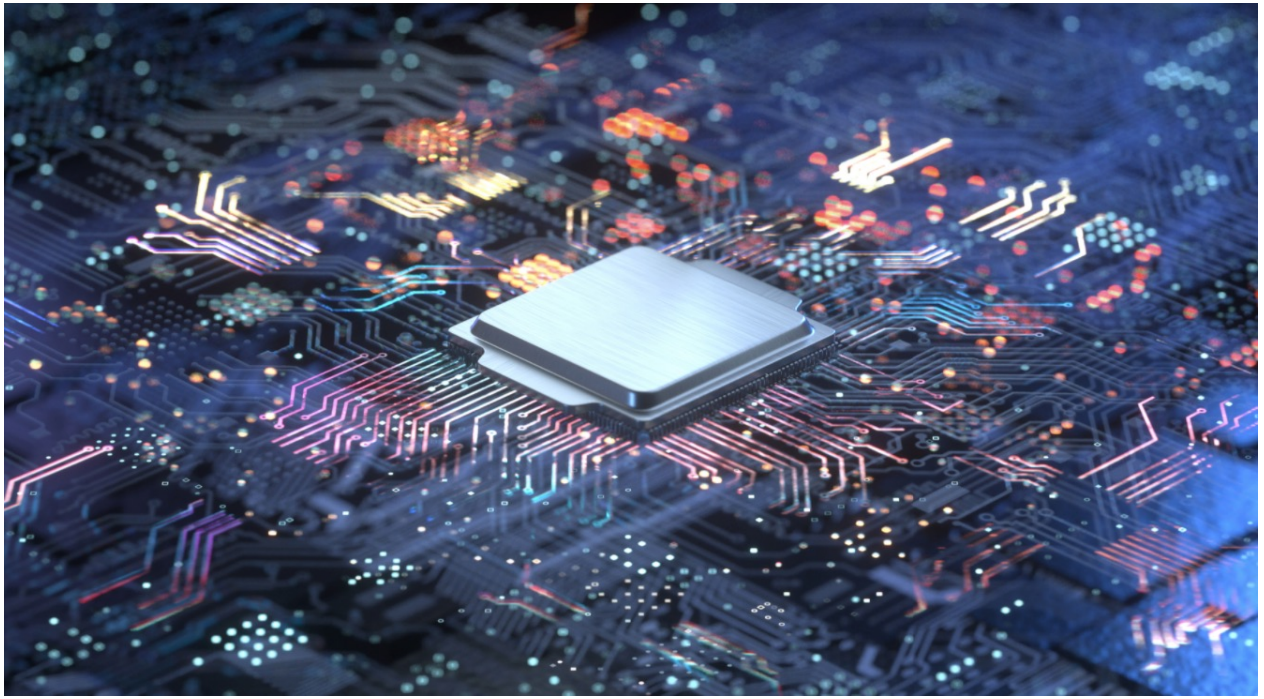
其他技术领域的“Native”

- 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

课堂思考

□讨论题

- 把一个传统的单体 Java Web 应用，原封不动地部署到阿里云的虚拟机上，这算不算“云原生”？为什么？

□上云 ≠ 云原生

- “云上的应用” 不一定是 “云原生应用”
- 云原生强调架构、流程、文化的整体变革，而不仅是部署位置

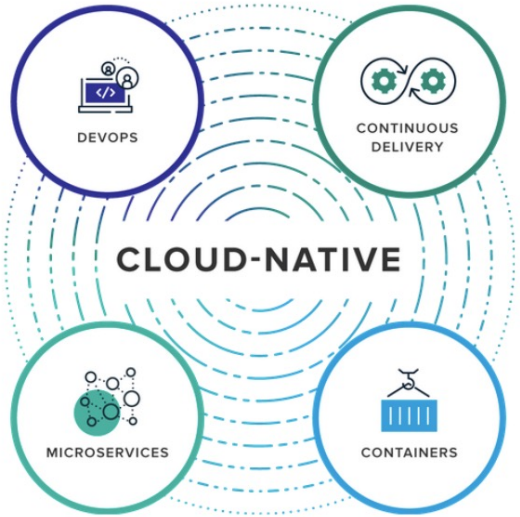
□判断是否“云原生”的关键问题

- 应用能否充分利用云的弹性扩缩容能力？
- 应用是否支持快速、频繁的迭代和发布？
- 应用是否容错？某个实例挂掉会不会影响整体服务？

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

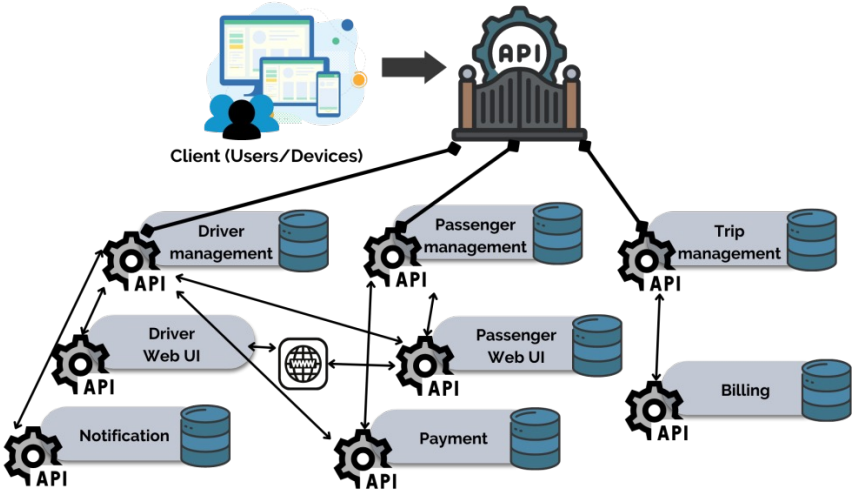


- **微服务**
  - 应用间通过 Restful API 通信
  - 可以被独立地部署、更新、扩容和重启
- **DevOps**
  - 自动化发布/部署流程
  - 快速部署到生产环境
  - 开发、运维协同合作
- **持续交付**
  - 频繁发布、快速交付、快速反馈、降低发布风险
- **容器化**
  - 微服务的最佳载体

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



案例分析：Netflix 的微服务之路

□背景：从 DVD 租赁到全球流媒体巨头

- 2008 年，Netflix 因数据库故障导致 DVD 业务中断 3 天
- 决定彻底重构：从单体架构迁移到微服务架构，全面上云（AWS）

□今天的 Netflix

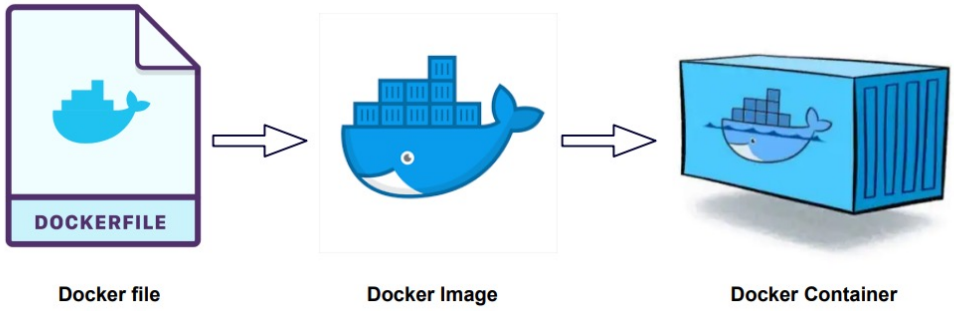
- 由 700+ 个微服务组成，每天处理上亿用户的视频请求
- 每天部署上千次代码，单个服务故障不影响整体观看体验
- 开源了 Eureka、Hystrix、Zuul 等一系列云原生中间件

□“混沌工程” Chaos Monkey

- Netflix 主动在生产环境随机杀掉服务实例，倒逼系统“为故障而设计”
- 启示：云原生应用必须假设“底层一定会出故障”

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

容器就像“集装箱”

□类比：从“散装运输”到“集装箱革命”

- 20 世纪 50 年代以前，海运货物各种各样，装卸耗时数天
- 集装箱（Container）标准化后，装卸时间缩短 90%，全球贸易爆发

□软件领域的同样变革：Docker 容器

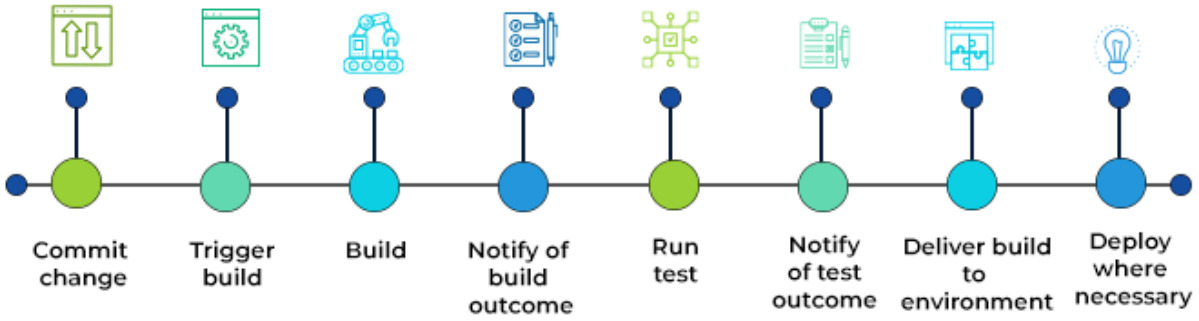
- 把应用 + 依赖 + 配置 打包成标准化的“集装箱”
- “一次构建，到处运行”——开发机、测试机、生产云上行为完全一致
- 启动以秒计，相比虚拟机的分钟级启动快 100 倍以上

□Docker 自 2013 年发布后迅速成为事实标准

- GitHub Star 数突破 6 万；Docker Hub 镜像下载量超过千亿次
- 彻底改变了软件交付方式，是云原生兴起的关键引擎

持续集成与持续交付技术

- **让软件从“写完代码”到“测试、构建、部署上线”的过程尽可能自动化、标准化、快速化**
  - **持续集成 (Continuous Integration, CI)**
    - 不要等到最后才集成，而是将代码频繁地集成到主分支中
    - 自动化的构建和测试，以尽早地暴露和修复问题，提高软件质量
  - **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（代码随时可发布）
    - 变更包括新功能开发、配置更改、Bug 修复等

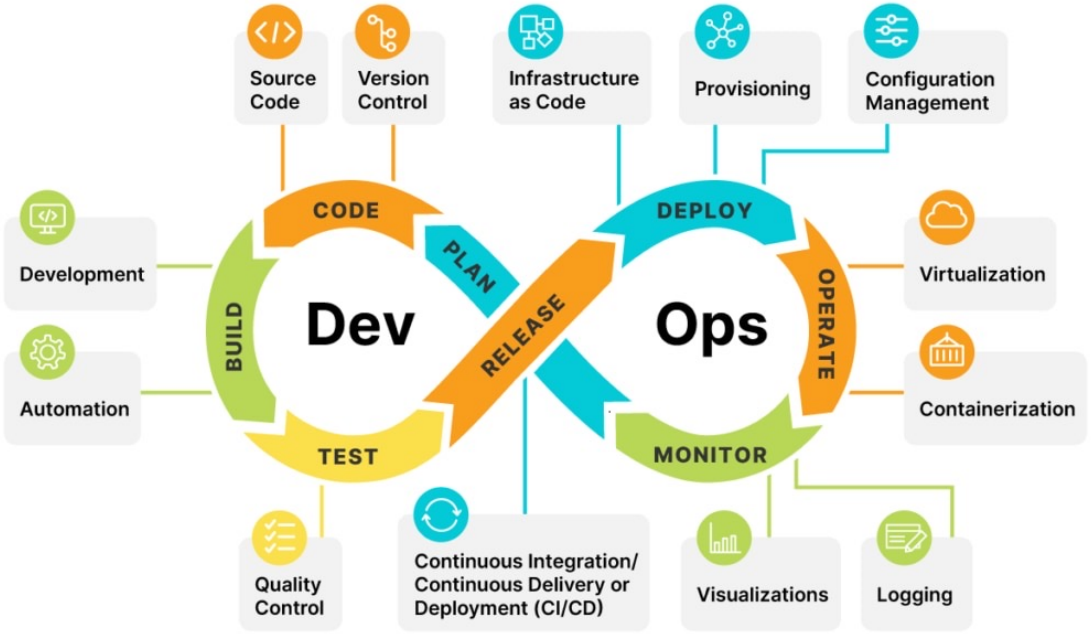


CI/CD Pipeline

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



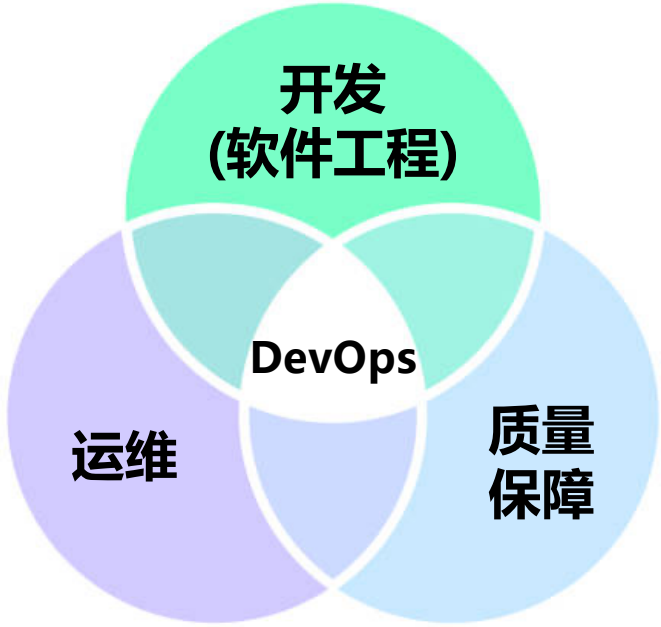
DevOps

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

谷歌前 CEO 埃里克·施密特

"Glue People" are also important



云原生归纳

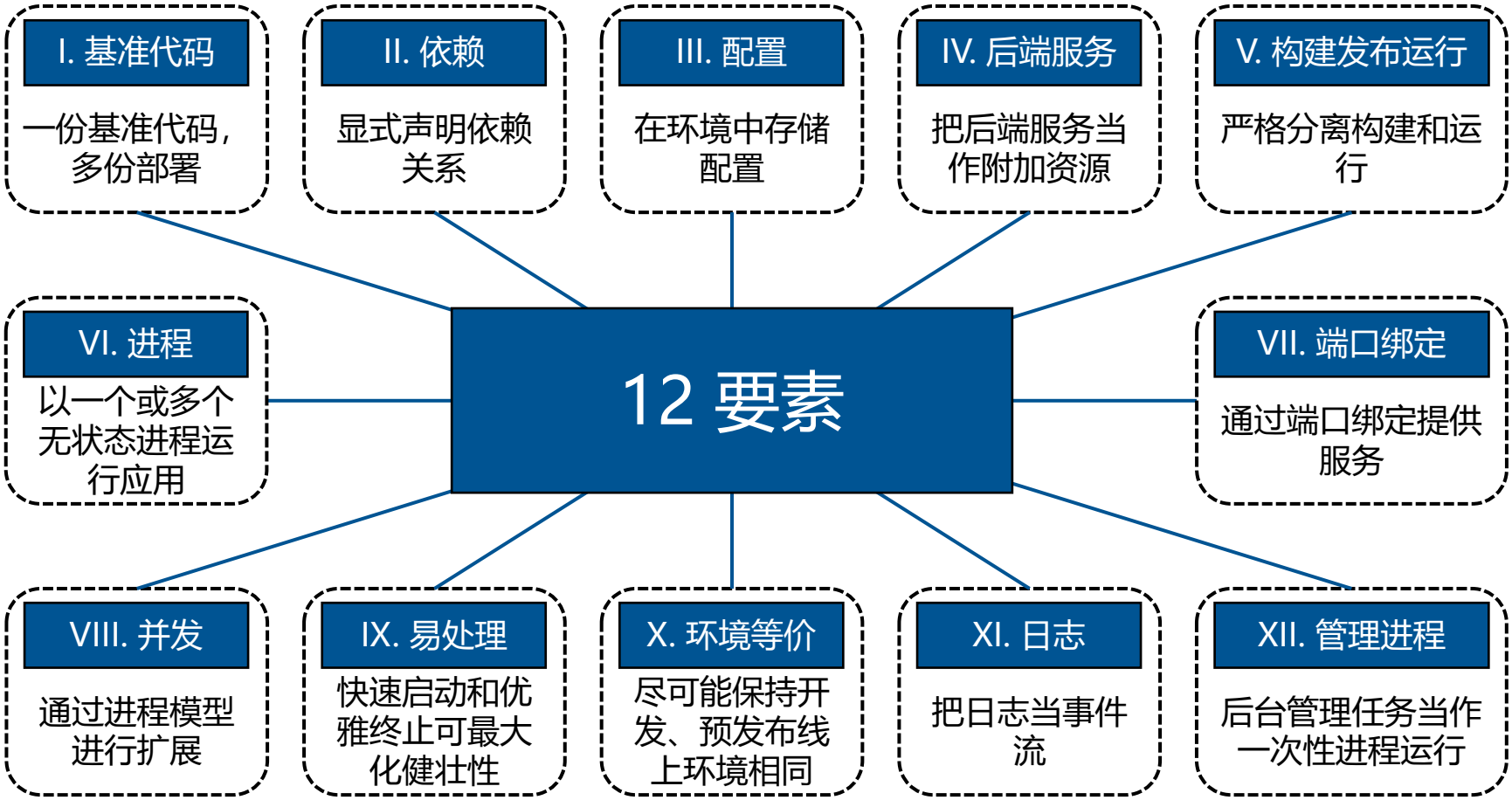
- 充分利用云计算技术的优势：采用**云端优先策略**，从云服务中获取最大价值
- 实现**快速、敏捷、频繁的交付模式**
- 通过技术创新更多地**扩展云计算技术的边界**

- 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一一对应的:
  - **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

- 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**



云原生应用开发实践的 12 要素

口一句话总结： 「让你的应用，天生就是为云而生」

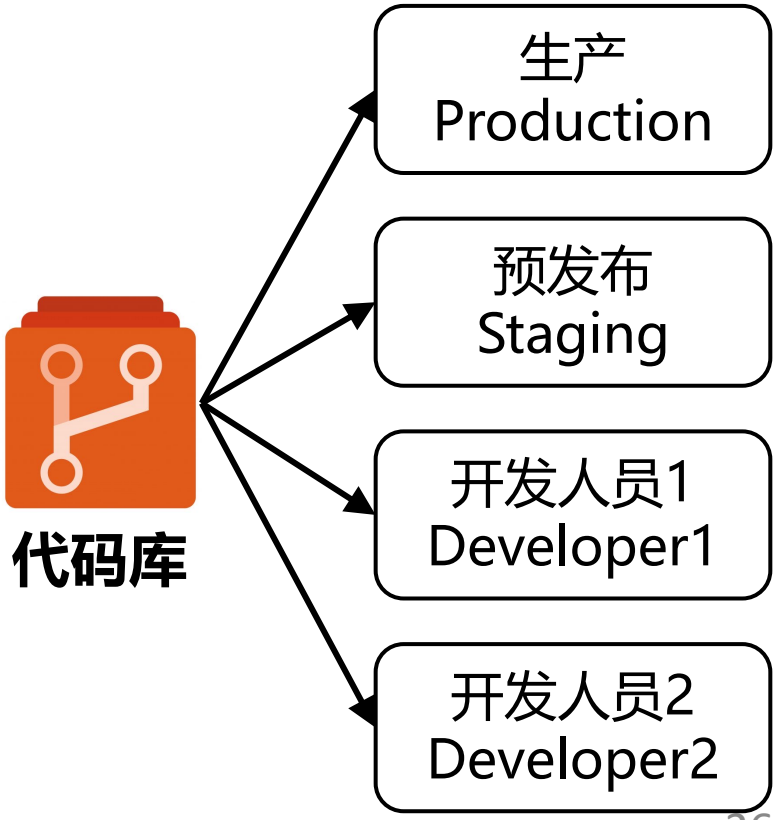
- **代码组织 (1-2)：** 代码库唯一 · 依赖显式声明
- **配置与依赖 (3-4)：** 配置外部化 · 后端服务即资源
- **构建与运行 (5-7)：** 构建/发布/运行三阶段分离 · 无状态 · 自带端口
- **扩展与运维 (8-10)：** 进程横向扩展 · 优雅启停 · 环境一致
- **可观测性 (11-12)：** 日志即事件流 · 管理任务即进程

不需要死记 12 条，记住每条对应的「生活类比」就够了！

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

- 🎯 类比：连锁店的「中央菜谱」。无论在广州还是上海开分店，都按总部的菜谱做菜；分店之间的差别只是「今天卖几份」，而不是菜谱本身。
- ❌ 反例：开发把代码拷贝到 U 盘改了改，直接放到生产服务器跑「线下定制版」。
- ✅ 正例：一个 Git 仓库 → 开发 / 测试 / 生产环境都从同一个仓库 checkout 不同版本（如 v1.2.3）部署。



原则 2：显式声明依赖关系

应用必须把它依赖的所有库都「写在清单上」，不能假设「系统里反正有」

- 🎯 类比：搬家清单。如果你不把所有家当列清楚，搬家公司到了新家就会发现「咦？锅怎么没带？」。
- ❌ 反例：代码里写了 import requests，但 requirements.txt 里没列，换台机器就跑不起来。
- ✅ 正例：requirements.txt / package.json / Dockerfile 完整列出依赖，docker build 后镜像「自包含」。

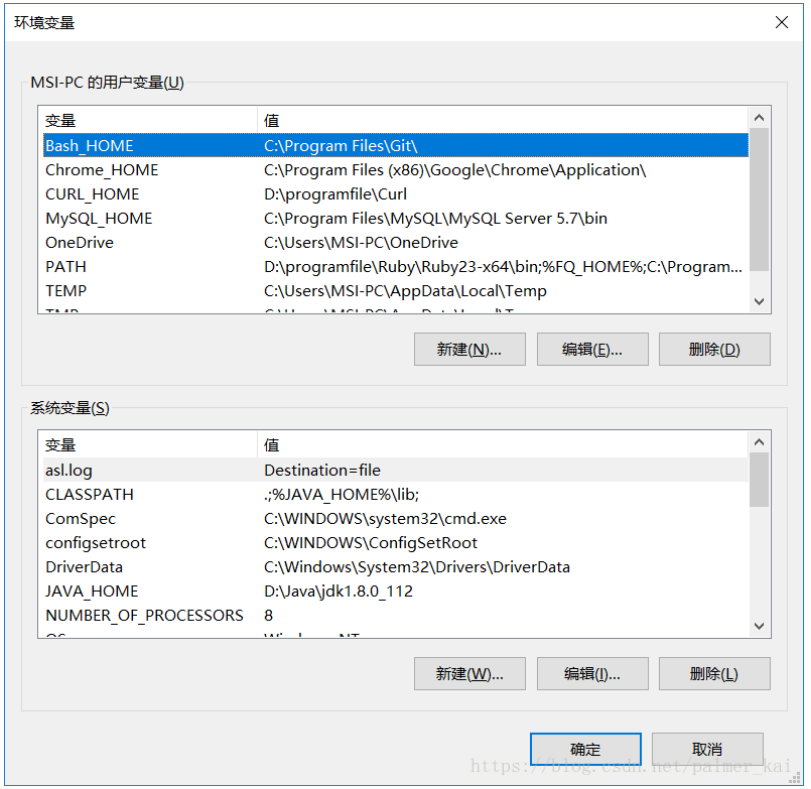
```
FROM golang:1.13 as builder
ENV PATH /go/bin:/usr/local/go/bin:$PATH
ENV GOPATH /go
COPY . /go/src/github.com/liqotech/liqo
WORKDIR /go/src/github.com/liqotech/liqo
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build ./cmd/crd-replicator/
RUN cp crd-replicator /usr/bin/crd-replicator

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

把会随环境变化的东西（数据库地址、密钥、端口）从代码里抽出来，通过环境变量注入

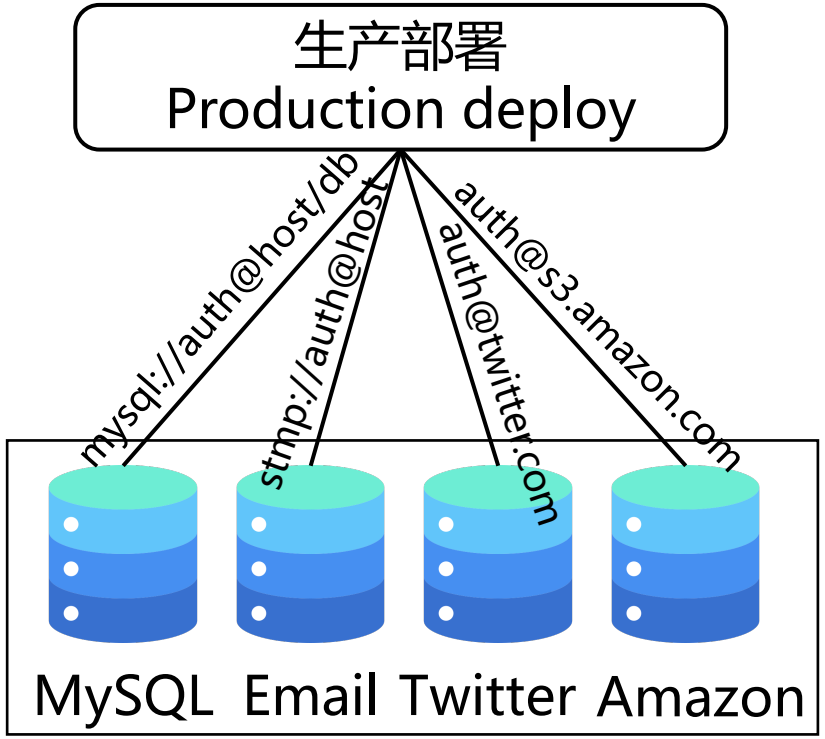
- 🎯 类比：演员的剧本（代码）不变，但每个剧院的舞美、灯光、音响（配置）不同，由现场环境提供。
- ❌ 反例：把 db_password = "123456" 写死在代码里，提交到 GitHub 后被爬虫扫到，数据库被脱裤。
- ✅ 正例：代码读取 os.environ['DATABASE_URL']，密码通过 K8s Secret 或环境变量传入。



原则 4：把后端服务当作附加资源

数据库、消息队列、缓存等后端服务，对应用来说都应该是「即插即用」的资源，通过配置切换

- 🎯 类比：家里用水用电。你不关心水是从哪个水库来的，水管接上就能用；换个房子，水管照样接，不用挖新井。
- ❌ 反例：代码里硬编码 jdbc:mysql://localhost:3306/mydb，迁到云上要改一堆代码。
- ✅ 正例：把数据库 URL 写进配置，从本地 MySQL 切到阿里云 RDS，应用代码一行都不用改。

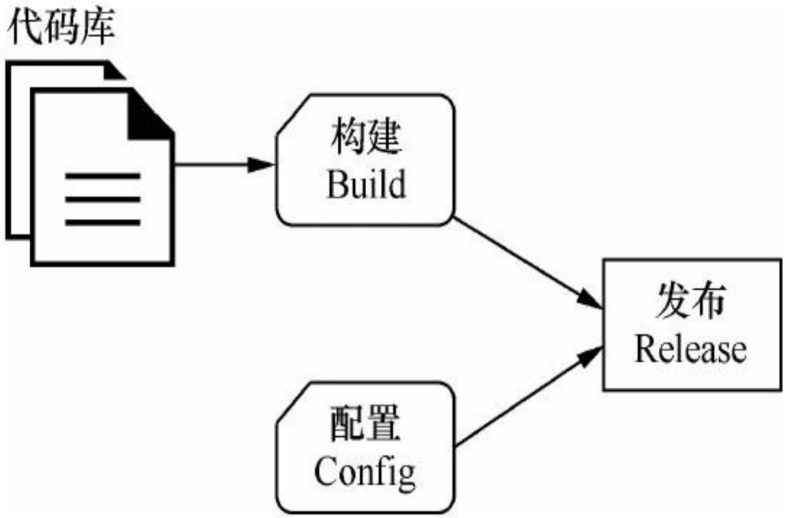


附加资源

原则 5：严格分离构建和运行

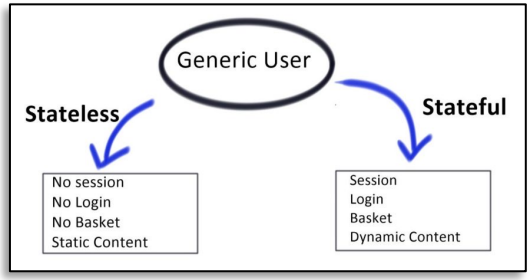
把「打包构建」和「上线运行」分成两个独立阶段，每次发布对应一个不可变的版本

- 🎯 类比：餐厅的「备菜」和「出餐」必须分开。顾客点单后，厨师不能现去市场买菜，必须早早把食材准备好。
- ❌ 反例：直接在生产服务器上 git pull 然后改两行代码再重启，「祖传热补丁」，没人知道线上跑的到底是哪个版本。
- ✅ 正例：CI 流水线构建出镜像 v1.2.3 → 推送到镜像仓库 → 生产环境拉取并运行；版本可追溯、可回滚。



原则6：以一个或多个无状态进程运行应用

应用进程本身不保存任何会话/数据状态；要持久化的东西交给外部数据库或缓存



🎯 类比：连锁奶茶店的店员。任何一个店员都能给任何一位顾客出单；不能说「我只服务昨天那个顾客，他的会员积分在我脑子里」。

❌ 反例：用户登录后，把 Session 存在进程内存的 HashMap 里，一重启全没了，扩容也用不上新实例。

✅ 正例：Session 存 Redis；任意实例都能处理任意请求，挂了直接拉起新实例，无缝替换。

✅ 无状态（Stateless）

- 每次请求自带所有上下文，进程本身不保留前一次请求的痕迹。
- 例子：静态网页、REST API、图片缩略图生成服务、纯计算函数
- **优点：任意实例可处理任意请求 → 容易横向扩展、容错、替换**

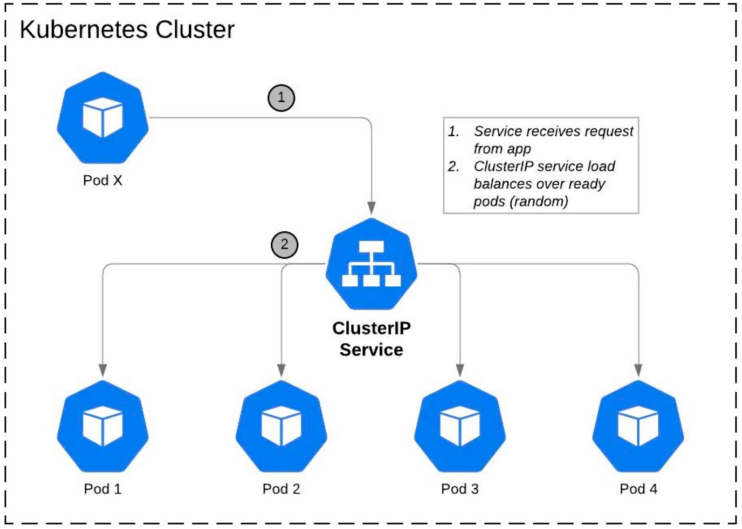
⚠️ 有状态（Stateful）

- 进程内部记着东西：会话、缓存、计数、累计数据……
- 例子：把用户登录信息存在内存里、本地计数器、未刷盘的数据库写入
- **问题：实例一挂数据就丢；新实例接不上原来的活；扩缩容很痛苦**

原则 7：通过端口绑定提供服务

应用自己监听端口对外提供服务，不依赖外部的 Web 容器（如 Tomcat）来托管

- 🎯 类比：从「商场里租柜台」到「自己开沿街店铺」。商场（Tomcat）关门你就没法做生意；自己的店铺，自己开门挂招牌。
- ❌ 反例：必须打包成 war 包，丢进 Tomcat 才能跑，部署强依赖外部容器。
- ✅ 正例：Spring Boot / Node.js / Go 应用 `java -jar` 或 `node server.js` 直接监听 8080 端口对外服务。



Kubernetes 服务完全符合此模式

原则 7：通过端口绑定提供服务

口传统 war 包 vs. 自包含应用

❌ 传统方式：寄生在 Tomcat

📦 **打包：myapp.war**
⬇️ 丢进 Tomcat 的 webapps/
🖥️ 启动 Tomcat（必须先装好）
⬇️
🌐 通过 Tomcat 间接对外服务

⚠️ **痛点：**
- 不能独立运行
- 部署前要装 Tomcat、配版本
- 一个 Tomcat 跑多个应用，互相影响
- Tomcat 升级 → 应用可能跑不起来

✅ 云原生方式：自包含应用

📦 **打包：app.jar / 单个二进制 / 镜像**
⬇️ 内嵌 Web 服务器（Tomcat / Netty）
🚀 一行命令启动
⬇️
🌐 自己监听端口，直接对外服务

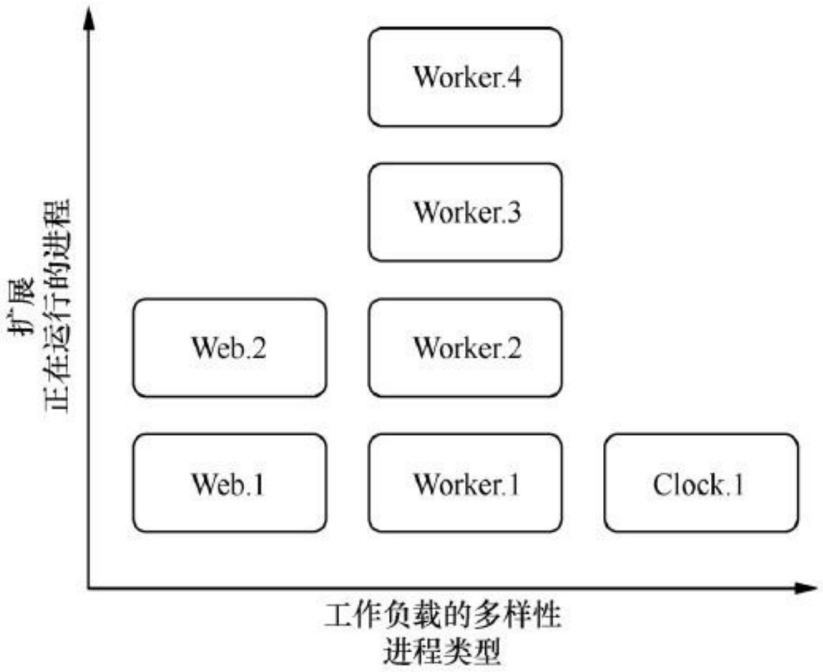
🎉 **优势：**
- 一行命令到处跑（笔记本/云/K8s）
- 完美适配 Docker：一容器一端口
- 微服务天然搭积木
- K8s Service 基于端口工作的基础

从「我的应用要部署到某个 Web 服务器」
→ 「我的应用本身就是一个服务」

原则 8：通过进程模型进行扩展

想扛更多流量？不是让单个进程变得更强，而是「多开几个进程」横向扩容

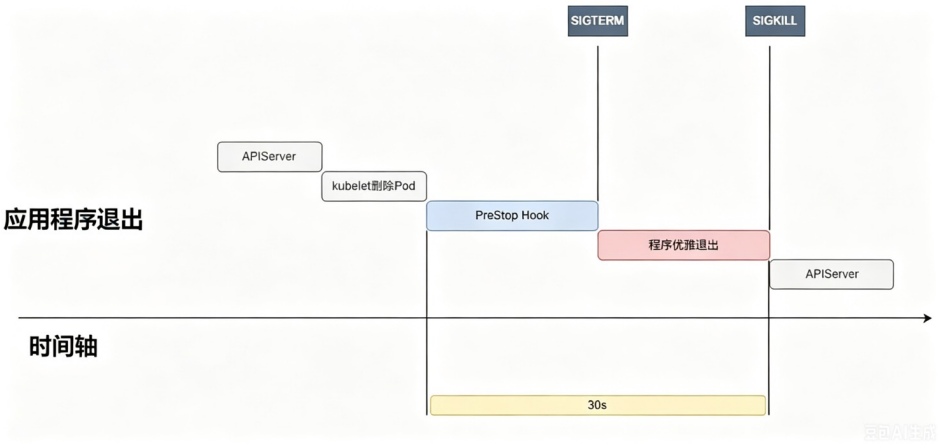
- 🎯 类比：餐厅高峰期人手不够，正确做法是再叫几个服务员（水平扩展），而不是让一个服务员练成「闪电侠」（垂直扩展）。
- ❌ 反例：单台服务器顶不住了 → 换更贵的 CPU、加更多内存，结果还是单点。
- ✅ 正例：把 Web 进程从 3 个扩到 30 个，前面挂个负载均衡，轻松扛住流量洪峰。



原则9：快速启动和优雅终止可最大化健壮性

应用要能秒级启动（应对扩容），收到停止信号时先处理完手头工作再退出（避免丢数据）

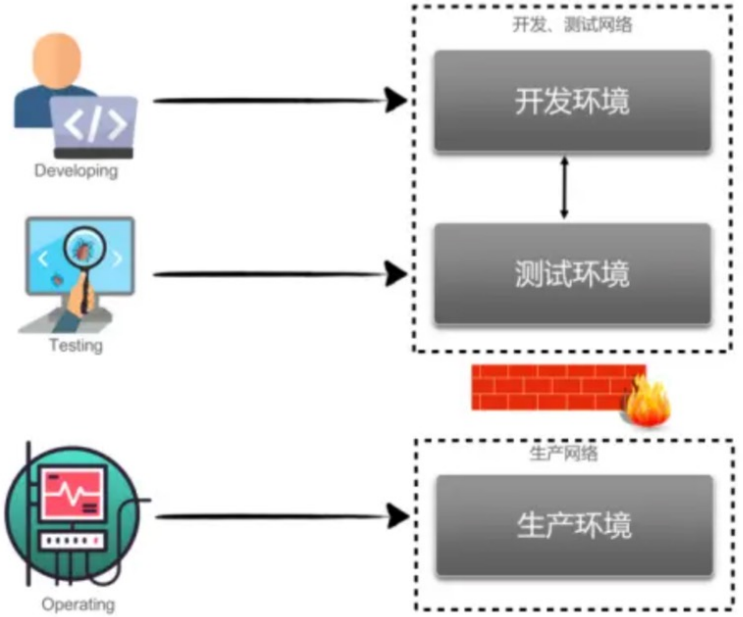
- 🎯 类比：消防员。听到警铃几秒钟出动（快速启动）；交接班时把正在救的火交代清楚再下班（优雅终止）。
- ❌ 反例：JVM 启动要 2 分钟才接请求 —— 大促扩容根本来不及；kill -9 强制结束，正在处理的订单全丢。
- ✅ 正例：容器秒级启动；监听 SIGTERM 信号，拒绝新请求 + 处理完当前请求 + 释放资源后退出。



原则 10：尽可能保持环境相同

开发、测试、生产三套环境的「时间差、人员差、工具差」越小越好，否则 Bug 永远抓不完

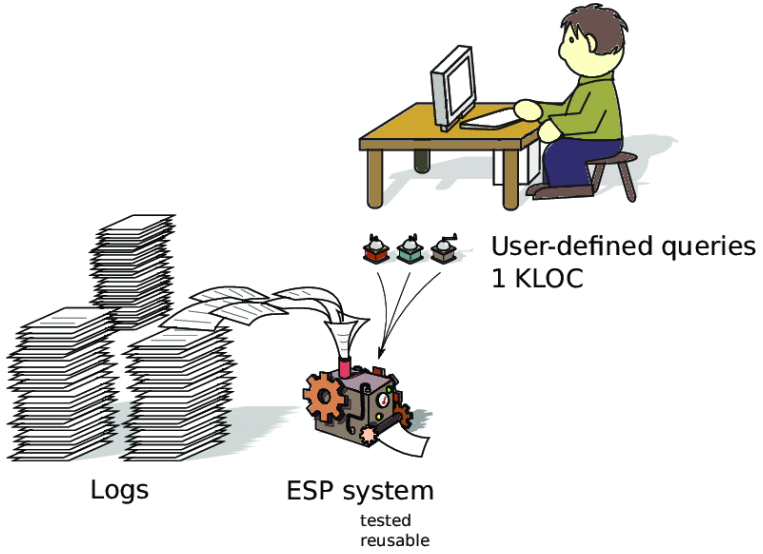
- 🎯 类比：足球比赛。如果训练场是泥地、比赛场是草地，球员到了赛场必然失常。
- ❌ 反例：开发用 macOS + SQLite，生产用 Linux + Oracle，「我本地是好的呀」成了团队口头禅。
- ✅ 正例：用 Docker 把整个运行环境固化下来，开发机跑的镜像和生产环境完全一致。



原则 11：把日志当作事件流

应用只管把日志「广播」到标准输出，怎么收集、存储、查询交给运行环境

- 🎯 类比：广播电台。主播只负责说话（应用打 log），听众用什么设备、录不录下来、存哪儿，是听众（运行环境）的事。
- ❌ 反例：应用自己写 logs/app.log 文件，容器一删，日志没了；多实例时日志散落各处，无法聚合。
- ✅ 正例：直接 print 到 stdout，由 K8s + Fluentd/Loki/ELK 集中采集，全集群日志统一搜索。



原则 12：后台管理任务当作一次性进程运行

数据库迁移、批量导数据、定时报表等运维任务，应该作为独立的一次性进程跑，跑完即退。

□🎯 类比：餐厅的「大扫除」请专门的清洁工，下班后或休息日来做，不影响日常营业；不会让正在炒菜的厨师同时拖地。

□❌ 反例：SSH 进生产容器里手动执行 SQL 迁移脚本，没有记录、没法回放、容易出错。

□✅ 正例：把迁移任务打包成 K8s Job 或 `python manage.py migrate`，与主应用共享同一份代码和环境，跑完自动退出。

进程隔离 → 不影响主业务

- 独立容器/进程，崩了不影响线上服务
- 占资源 / 跑超时 → 直接杀掉，主业务无感
- 退出码、日志独立，方便定位问题

跑完即退 → 天然云友好

- 不占用常驻资源，省钱省心
- K8s 可调度到任意空闲节点
- 完美呼应原则 6（无状态）和原则 9（快速启停）

课堂小测：12 要素你掌握了吗？

□判断下列做法是否符合 12 要素原则，并指出违反了哪一条

① 把数据库密码直接写在 application.properties 中提交到 Git
② 应用启动时把所有用户会话存到本地内存里的 HashMap
③ 一个项目的开发用 SQLite，生产用 Oracle，且经常出现差异 Bug
④ 应用直接把日志写到本地文件 /var/log/app.log
⑤ 收到 SIGTERM 信号后立即退出，丢弃所有正在处理的请求

□参考答案

① 违反原则 3（配置存于环境）
② 违反原则 6（无状态进程）
③ 违反原则 10（环境等同）
④ 违反原则 11（日志即事件流）
⑤ 违反原则 9（快速启动 & 优雅终止）

云原生的落地：Kubernetes

Kubernetes（K8s）

K8s 是云原生时代的「操作系统」，让你像管理一台超级电脑一样管理整个集群

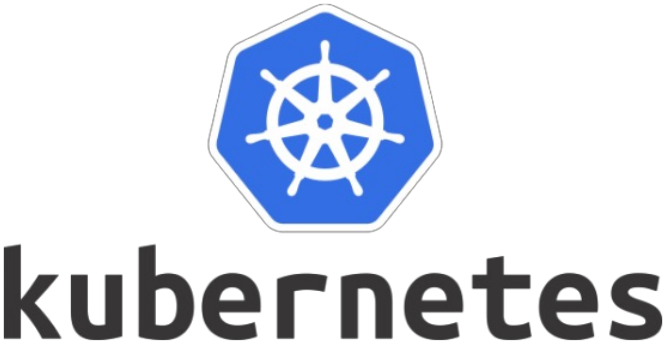
- **把 K8s 想象成一座「智能化集装箱码头」：**
  - 一边是源源不断到港的货物（你的容器），
  - 一边是无数个堆场（服务器），
  - K8s = 这座码头的调度中心，自动决定谁放哪、谁挂了换新的、流量怎么走。

- 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)，很快成为行业标准

Kubernetes (K8s)

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

- K8s 的主要功能包括：
  - 自动部署和回滚
  - 服务发现和负载均衡
  - 存储管理
  - 容器健康管理
  - 密钥和配置管理



How to pronounce Kubernetes?

coo-ber-net-ees

(even when it is shortened as K8s)

Kubernetes 到底帮你做了什么？

如果没有 K8s，你想要部署 100 个容器，
得自己操心这些事——

- **自动部署 & 回滚：**新版本一键灰度上线，出问题一键回滚到旧版本
- **弹性扩缩容：**流量来了自动多开容器，流量走了自动关掉，省钱省心
- **健康检查 & 自愈：**容器挂了？K8s 自动重启或换一台机器重新拉起
- **服务发现 & 负载均衡：**100 个相同容器对外只暴露 1 个稳定地址，请求自动分发
- **存储编排：**容器是临时的，但数据要持久化 —— K8s 帮你挂载云盘
- **配置 & 密钥管理：**配置和密码集中管理，按需注入到容器里，再也不用硬编码

案例分析：Pokémon Go 与 Kubernetes

□2016 年现象级游戏 Pokémon Go 上线

- 预计用户量 500 万，实际峰值用户量超过 5000 万 —— 是预期的 10 倍！
- 如果用传统架构，必然崩溃数周；但游戏只在前几天有短暂宕机

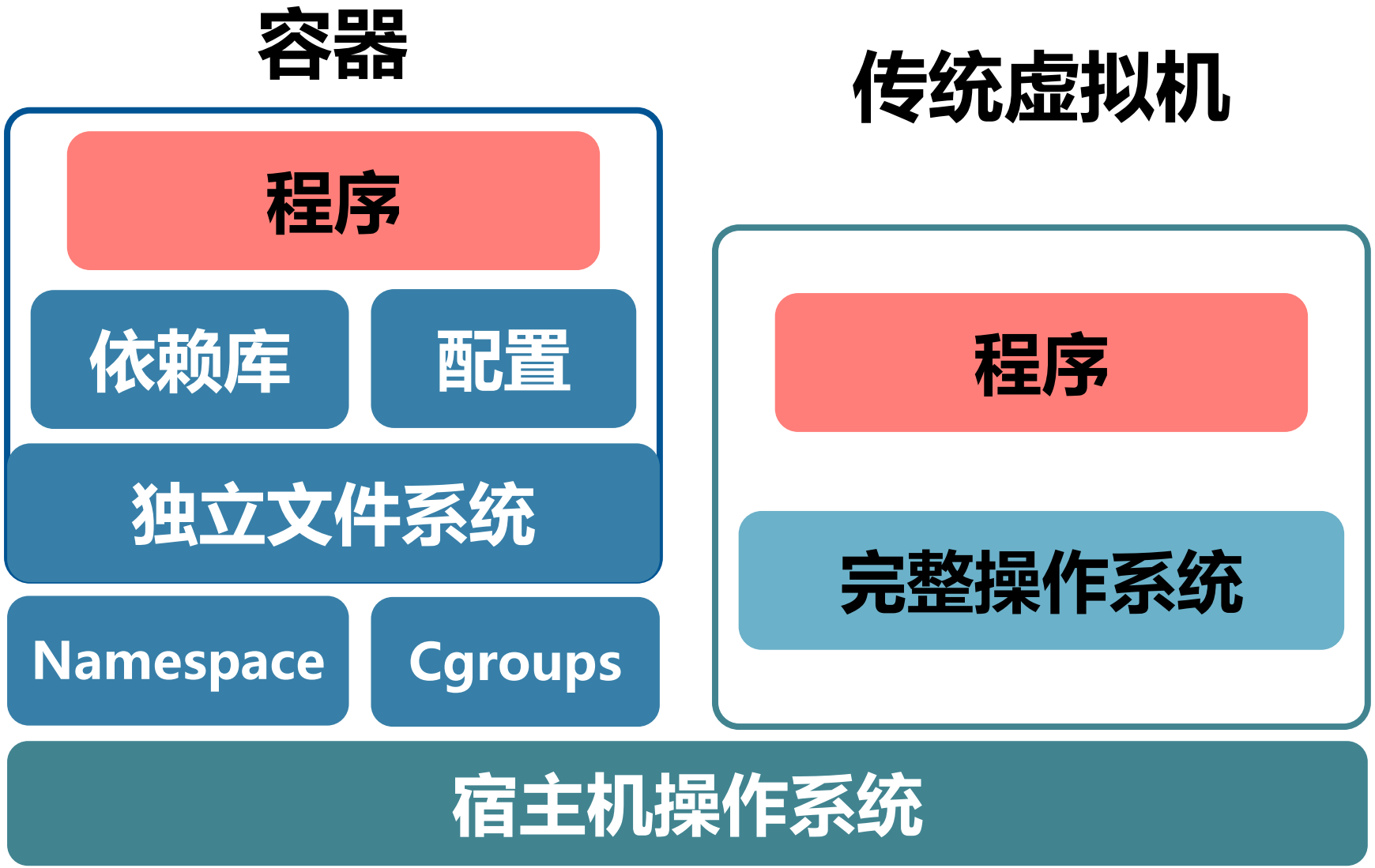
□成功的秘密：Google Cloud 上的 Kubernetes

- 游戏后端跑在 GKE（Google Kubernetes Engine）上
- K8s 在流量飙升时自动扩容数千个 Pod，平稳应对全球玩家涌入
- Google SRE 团队称之为 “GKE 史上最大规模的部署案例之一”

□国内案例：阿里巴巴双 11

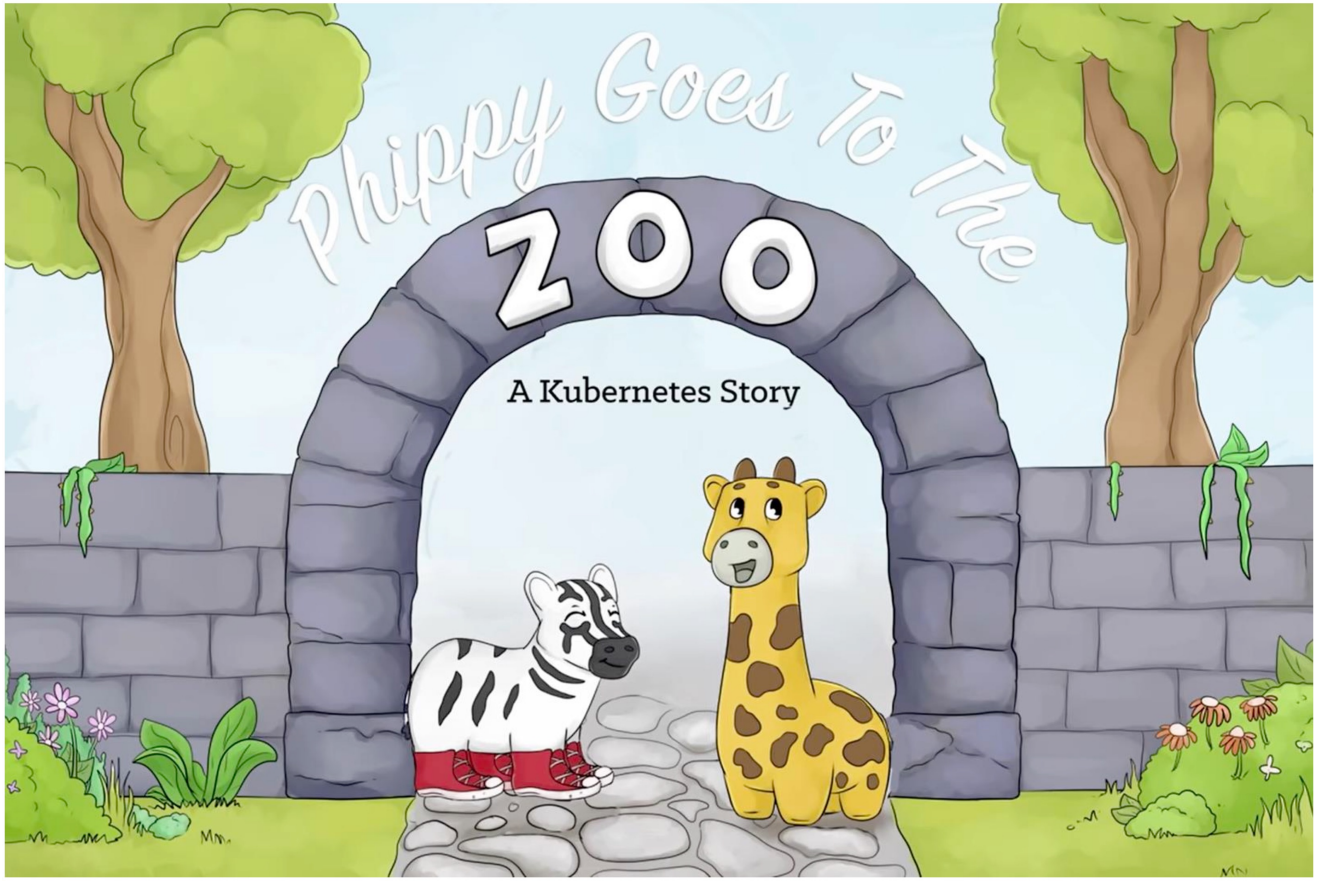
- 基于 Kubernetes 管理百万级容器，支撑每秒数十万笔交易
- 云原生让“大促”从“熬夜抢救”变成“点几下扩容按钮”

容器回顾


容器
程序
依赖库
配置
独立文件系统
Namespace
Cgroups
传统虚拟机
程序
完整操作系统
宿主机操作系统

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

- 你只跟「门牌号」打交道，背后是阿里云、AWS、还是自建机房，K8s 帮你屏蔽（通过 Kubernetes 提供的统一 API 描述“我要什么”）
- 就像你点外卖只关心「30 分钟送到」，不关心骑手骑什么车

□声明式配置 (Declarative configuration)

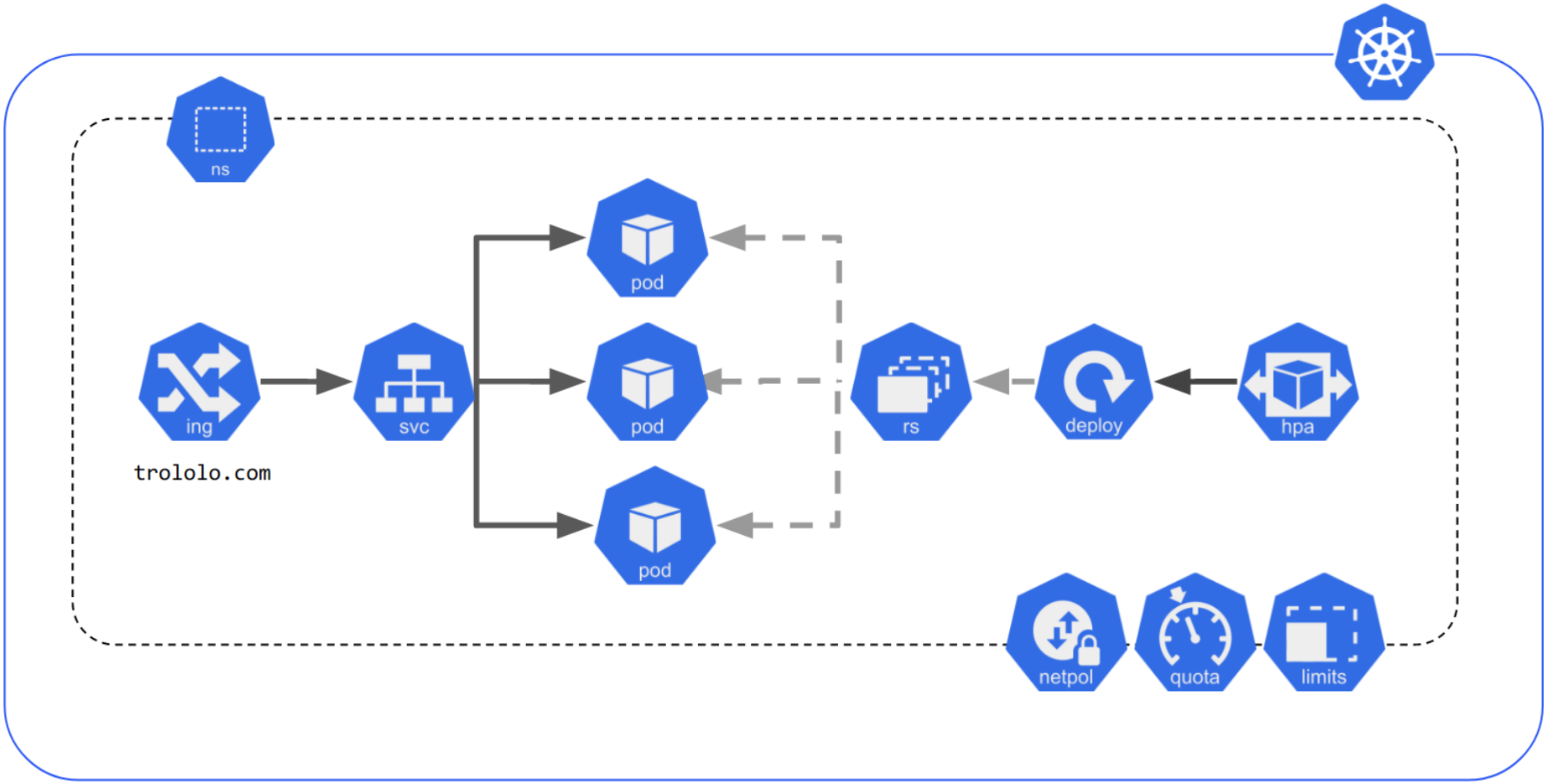
- 你只告诉系统「我要什么样」，不告诉它「怎么做」
- 比如：「我要 3 个 Pod 始终在跑」—— 至于怎么调度、宕机怎么补救，K8s 自己想办法

□以控制循环 (Control Loop)

- K8s 不停地问自己：「现在的状态」 = 「期望的状态」吗？不一样就动手调整
- 像家里的空调：你设定 26°C，温度高了它制冷、低了它停下，永远朝目标收敛

Basic Kubernetes Objects

口基本 Kubernetes 对象



K8s 对象：用 YAML 描述你的「期望」

□一切皆对象，你写一份 YAML 描述目标，K8s 想办法实现

□每个对象都长这样：

```
apiVersion: v1          # 用哪个版本的 API
kind: Pod               # 我是什么类型（Pod / Service / Deployment ...）
metadata:
  name: my-web          # 名字
  labels: {app: web}    # 标签（给同伴们做识别）
spec:                   # ⭐ 我「期望」的状态：用户来填
  containers:
    - image: nginx:1.25
status:                 # 当前的实际状态：K8s 自动更新
  phase: Running
```

□可以理解为：你写「订餐单」描述菜品（spec），餐厅写「出餐进度」回执（status）

Pod：K8s 世界的最小公民

□Pod 是 K8s 能调度的最小单位 —— 不是单个容器！

- **类比码头：** Pod = 用绑带捆在一起、必须一起上下船的几个集装箱
- 它们共享同一辆运输车（节点）、同一个网络（IP/localhost）、同一个仓位（数据卷）

□Pod 内部容器有什么共享?

- 共享网络命名空间 —— 互相通过 localhost 直接通信
- 共享存储卷 —— 可以读写同一份文件
- 共同生死 —— 一起被调度、一起被销毁

□注意：Pod 是「一次性」的，挂了不会原地重生，K8s 会启一个新的（IP 都变了）

Pod

- **90% 的场景：一个 Pod 里只跑一个容器**
- **那为什么 Pod 还允许多容器?**
- 应用场景叫做 **「Sidecar 模式」**：主容器旁边带一个「跟班」
  - 主容器是 web 应用，Sidecar 帮它收集日志/采集监控指标
  - 主容器是业务代码，Sidecar 做服务网格代理（如 Istio）
- **Pod 之于 K8s，相当于「进程」之于操作系统**

```
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：给一群 Pod 起个稳定的门牌号

□Pod 的 IP 会变，但 Service 的地址永远稳定

□类比：奶茶店总部对外只公布 「400-客服热线」（Service）

- 具体哪个店员（Pod）接电话、是不是换班了，顾客不关心
- 比如user-service、order-service 、payment-service

□使用方用 **http://my-service:8080** 这种域名访问，不用关心后面有几个 Pod、它们在哪台机器

□Service 干两件大事：

① **服务发现：**Pod 新增、删除、被换到别的节点，Service 自动跟踪，使用方无感知

② **负载均衡：**把流量平均分给后端的多个 Pod，单个 Pod 挂掉自动剔除

Service 的三种「开放程度」

□类比：办公楼的三种门禁

□ClusterIP（默认）

- 内部专用：只有「楼里员工」（集群内的 Pod）能访问
- 适合：微服务之间互相调用

□NodePort

- 每个节点都开一个固定的「侧门」（端口），外部可以从任一节点进入
- 适合：调试、内网访问，不推荐生产暴露

□LoadBalancer

- 门口请来云厂商的「专业保安」（云负载均衡器）统一接待外部用户
- 适合：生产环境对外暴露，自动拿到公网 IP

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：给 Pod 配一个「行李箱」

- **容器是「一次性」的，但你的数据需要保存下去**
- **问题：**如果容器挂了重启，里面写入的文件全没了，怎么办？
- **解法：Volume（数据卷）**
  - 类比：每个 Pod 出门时带一个「行李箱」，里面装的东西不会随着容器销毁而消失
  - Volume 挂载在 Pod 上，Pod 里的多个容器都能读写
- **常见后端：**
  - emptyDir：临时空间，Pod 死了也跟着没（适合容器间交换数据）
  - hostPath：直接挂主机目录（适合日志收集）
  - PVC：动态申请云盘/NFS，Pod 死了数据还在（生产环境标配）

Namespace：一个集群里划「楼层」

□把一个物理集群，逻辑上分成多个互不打扰的「小集群」

□类比：写字楼按楼层划分

- 同一栋楼（物理集群），3 楼是研发部，5 楼是财务部，
- 彼此之间互相隔离 —— 各楼层有自己的会议室预算、门禁权限、保洁安排

□Namespace 给你三大好处：

- 资源配额：限制某个团队最多用多少 CPU/内存
- 权限隔离：开发人员只能动开发命名空间的资源
- 名字解耦：dev/test/prod 各自有 my-service，互不冲突

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

Controller：永不下班的「自动巡检员」

□控制器是 K8s 的灵魂 —— 它让集群「自愈」

□类比：商场里的自动巡检员

- 每隔几秒巡视一圈：「现在的样子」 vs 「应该的样子」
- 发现不一致 → 立刻动手修正，永不下班、永不疲倦

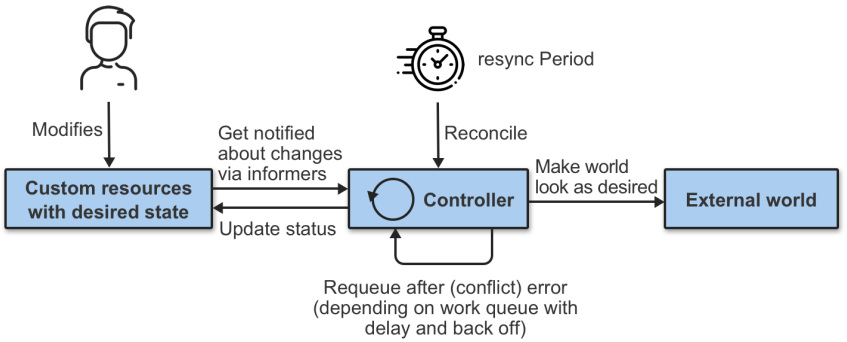
□控制循环（Control Loop）

while True:

current = 观察当前状态

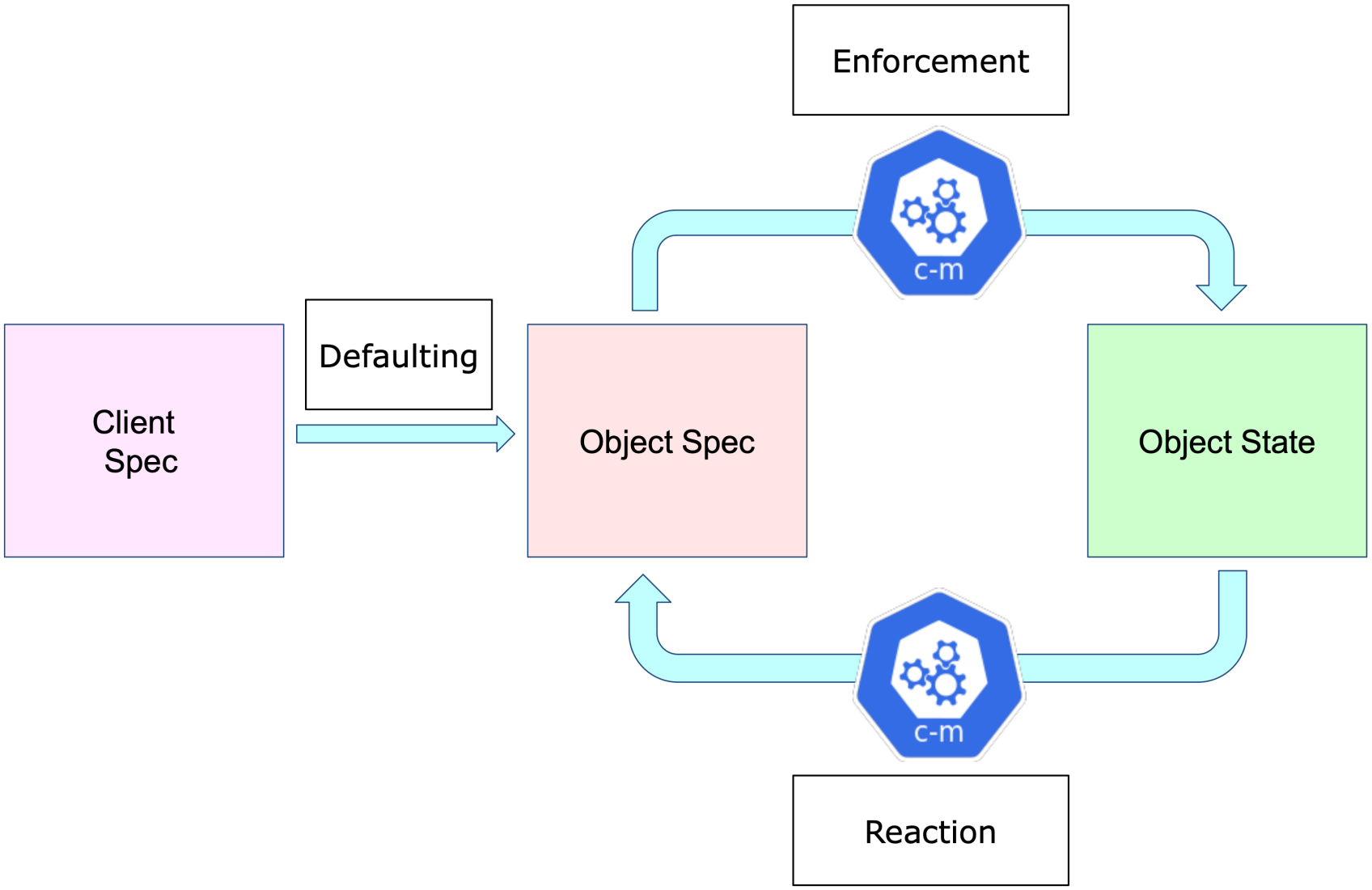
desired = 读取期望状态

if current ≠ desired: 采取行动



□接下来介绍几个常见的控制器： ReplicaSet、Deployment、HPA、DaemonSet、StatefulSet、CronJob

一种面向状态的方法


Enforcement
c-m
Client
Spec
Defaulting
Object Spec
Object State
c-m
Reaction

ReplicaSet：保持「副本数」

- **你说要 3 个 Pod，它就保证永远有 3 个，多了砍、少了补**
- 类比：连锁奶茶店总部规定「每家分店必须 3 个店员在岗」，有人请假？立刻补一个；招多了？立刻调走
  - 通过 labels/selector 锁定要管理的 Pod 集合
  - 节点挂了？在别的节点起新的副本，对外完全无感
  - 实际开发中很少手写 ReplicaSet，Deployment 会帮你管它

```yaml
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：日常最常用的「应用管家」

□管理应用的版本与发布，滚动升级、灰度、回滚，一行命令搞定

□类比：餐厅换菜单。Deployment 帮你「**逐桌**换菜」（滚动升级），新菜评价差？一键回到旧菜单（回滚）

- 三层关系：Deployment 创建 ReplicaSet，ReplicaSet 管理 Pod
- 一行命令滚动升级：kubectl set image deploy/web web=nginx:1.26
- 一行命令秒级回滚：kubectl rollout undo deploy/web
- 99% 的无状态应用都用 Deployment 部署

```yaml
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
```

HPA：根据负载自动「招人 / 裁员」

- **Pod 副本数不该写死，让它跟着 CPU、内存、QPS 自动伸缩**
- 类比：商场客流高峰自动叫更多保安，半夜没人就只留两个值班，永远按需配人
  - 设定范围：min=2, max=20 （守住下限，控制上限）
  - 设定目标：CPU 平均利用率 ≤ 70% （超过就扩，远低于就缩）
  - 指标来源：CPU/内存（默认），也可对接 Prometheus 的业务指标
- **把「值班表」从「固定人数」变成「按需自动算」**

```yaml
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：每个节点都派驻一名「楼层管家」

□确保每台机器上都跑一份 Pod，没有就补，节点退役就回收

□类比：写字楼里每层都得有一个保洁员；新开一层楼，自动派一个保洁过去

- 典型用途 1：日志收集器（Fluentd / Filebeat），每台机器各装一份
- 典型用途 2：监控采集（node-exporter），必须每台机器一份
- 典型用途 3：网络/存储插件（CNI / CSI），节点级基础设施

StatefulSet：为「有身份」的 Pod 而生

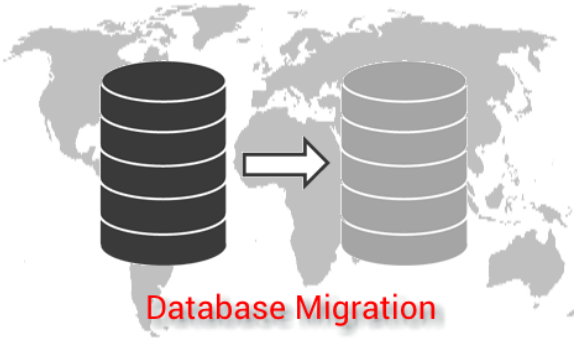
□数据库、ZooKeeper 这类有状态、有数据服务，Pod 不能随便换，名字、存储、启动顺序都要稳定

□核心理解：

- Deployment 适合无状态应用：Pod 坏了可以随便换
- StatefulSet 适合有状态应用：Pod 坏了要带着原身份、原数据回来

□StatefulSet 提供三种稳定性：

- 稳定名字：mysql-0、mysql-1、mysql-2
- 稳定存储：每个 Pod 绑定自己的持久化数据卷
- 稳定顺序：按 0→1→2 启动，按 2→1→0 删除

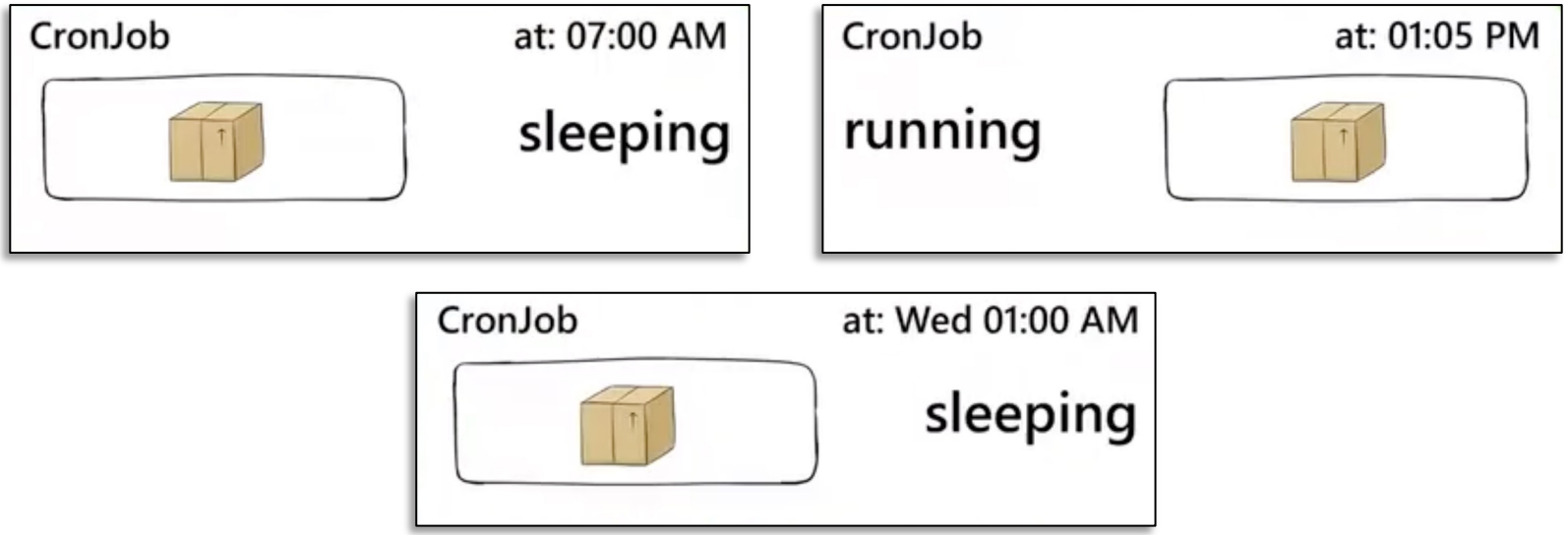


CronJob：K8s 里的「定时闹钟」

□按 cron 表达式定时启动一个一次性任务，跑完即退

□类比：每天凌晨 3 点叫醒清洁工大扫除；每周日发周报

- 0 3 * * *　每天凌晨 3 点跑数据库备份
- */10 * * * *　每 10 分钟跑一次报表同步
- CronJob → 触发 Job → 启动 Pod → 跑完销毁



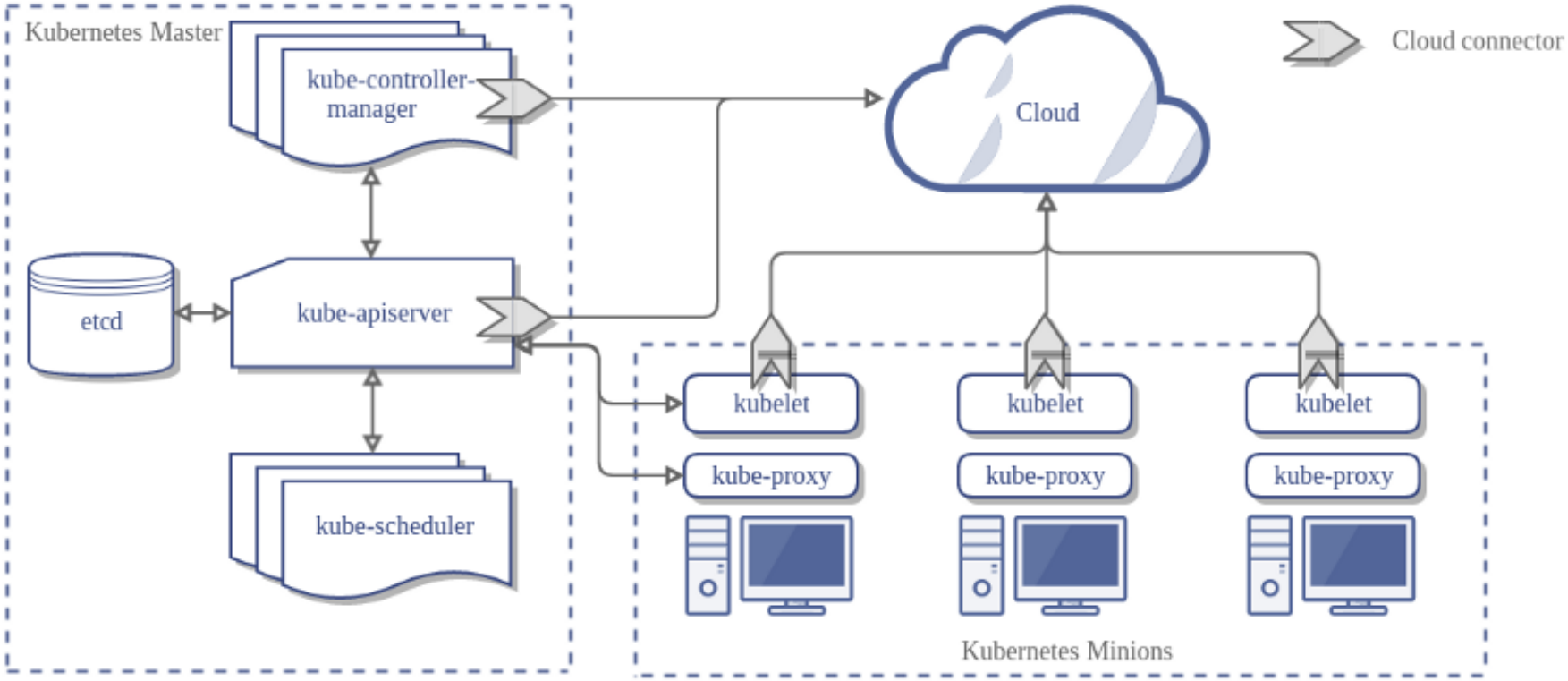
ConfigMap & Secrets：配置和密码住「集中宿舍」

□把配置从代码里剥离，集中保管；要用时注入到 Pod 里

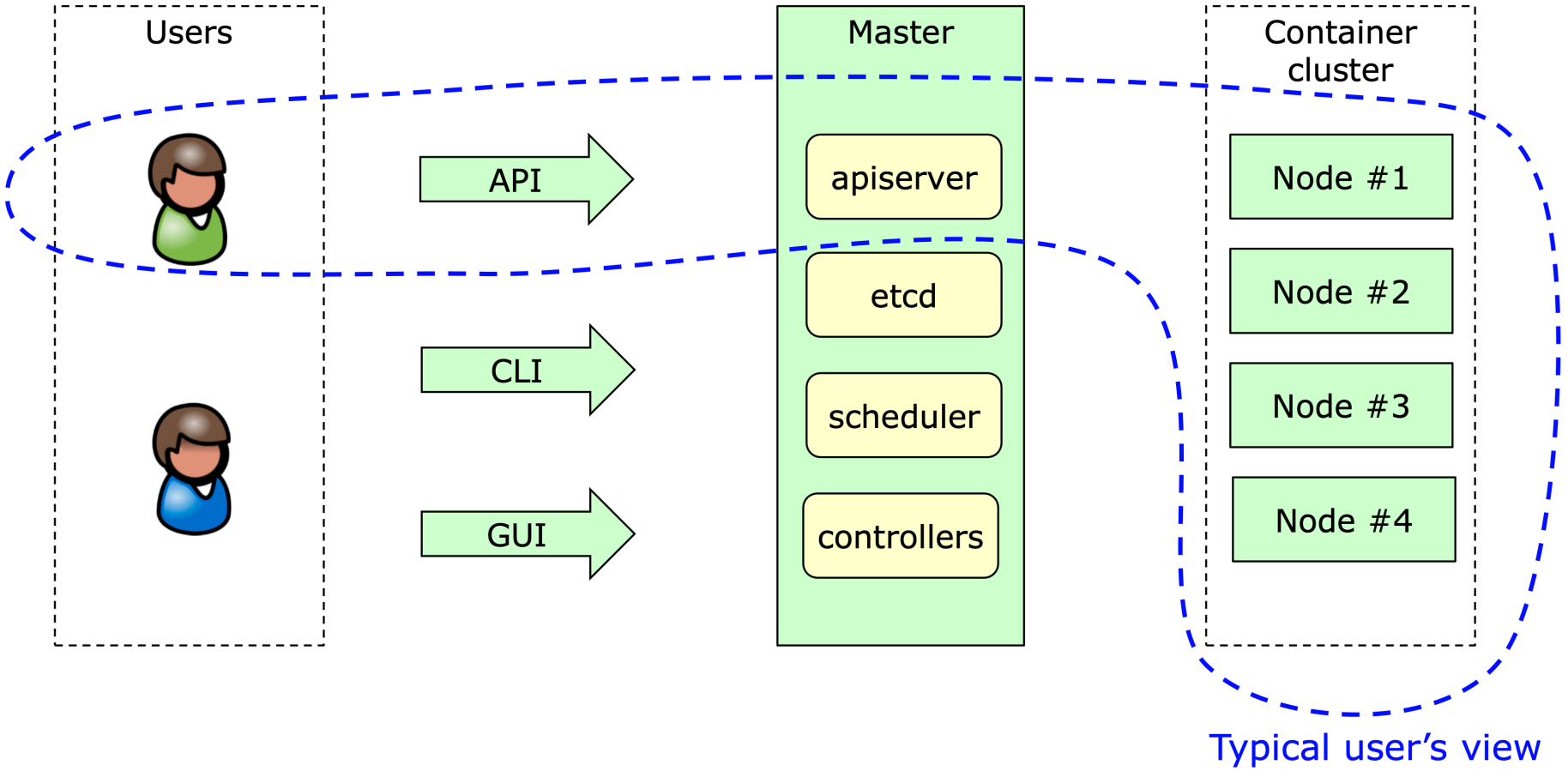
□类比：员工的工卡（ConfigMap：基本信息）和保险柜里的钥匙（Secrets：密码、令牌），都不和员工的衣服（代码）绑死

- ConfigMap：明文 KV，存普通配置（如 LOG_LEVEL=debug）
- Secrets：加密 KV，存敏感信息（密码、API Key、TLS 证书）
- 注入方式：环境变量 / 挂载成文件，应用代码无感知
- ✅ 完美呼应 12 要素的「原则 3：配置存于环境」

Kubernetes 软件架构

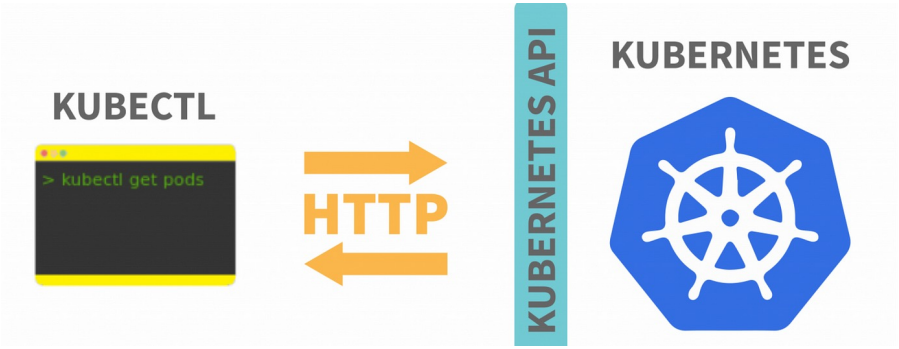


Kubernetes 软件架构：用户视角



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

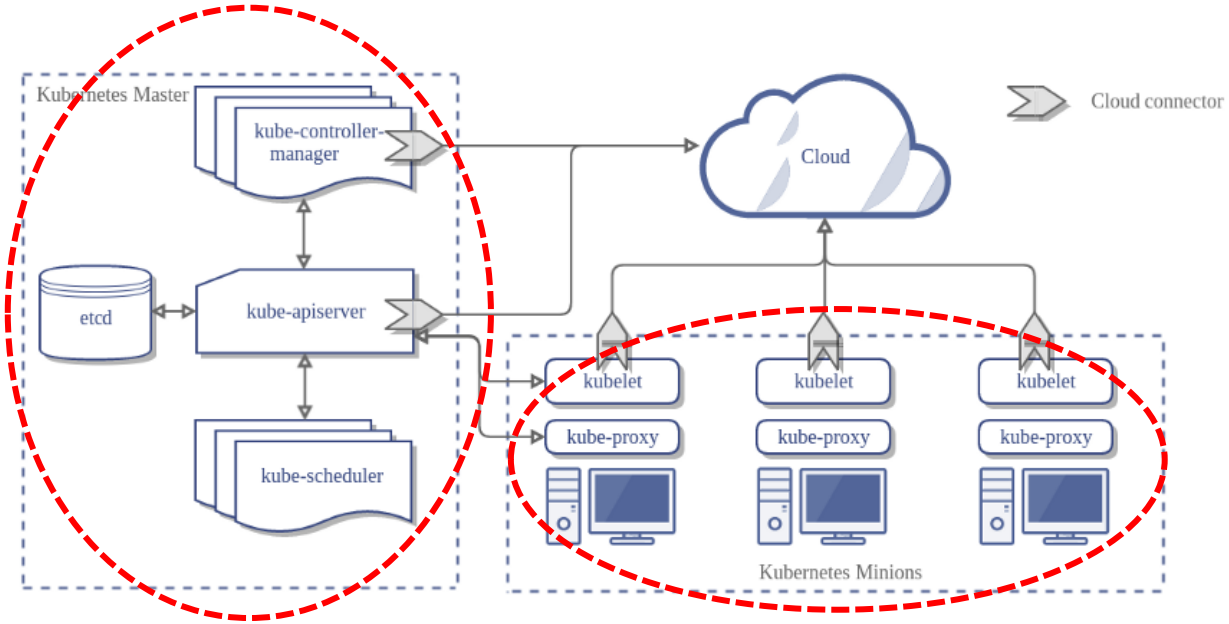


- Kubectl 的使用方式非常灵活
  - 在命令行中输入命令，如“kubectl create“创建一个新的 Pod
  - 用 YAML 或 JSON 文件来描述资源对象，然后用“kubectl apply“命令将其应用到集群中

软件架构：概述

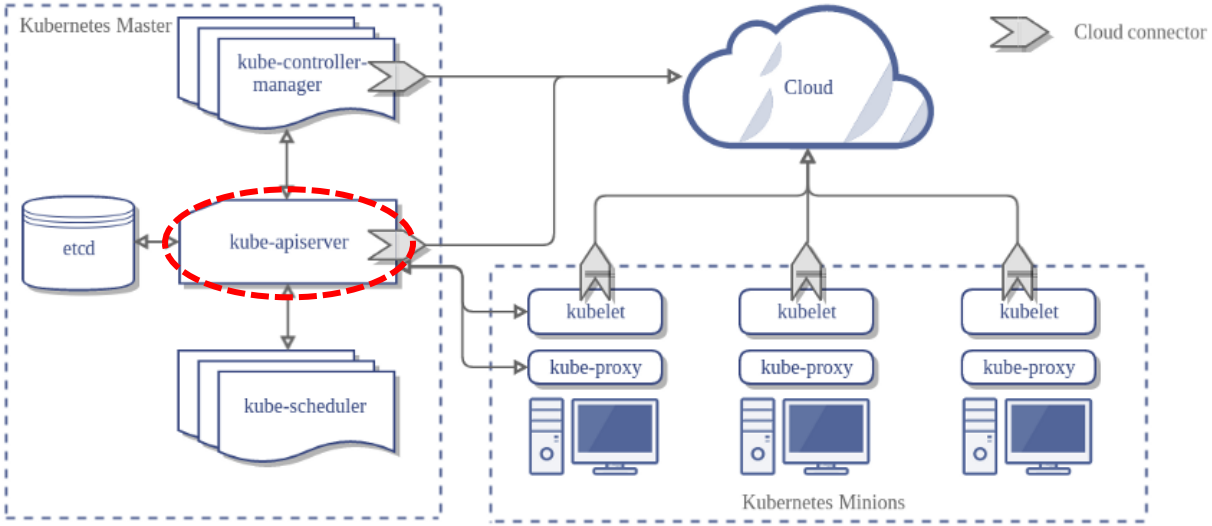
□在 K8s 中，主要有两类组件

- **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



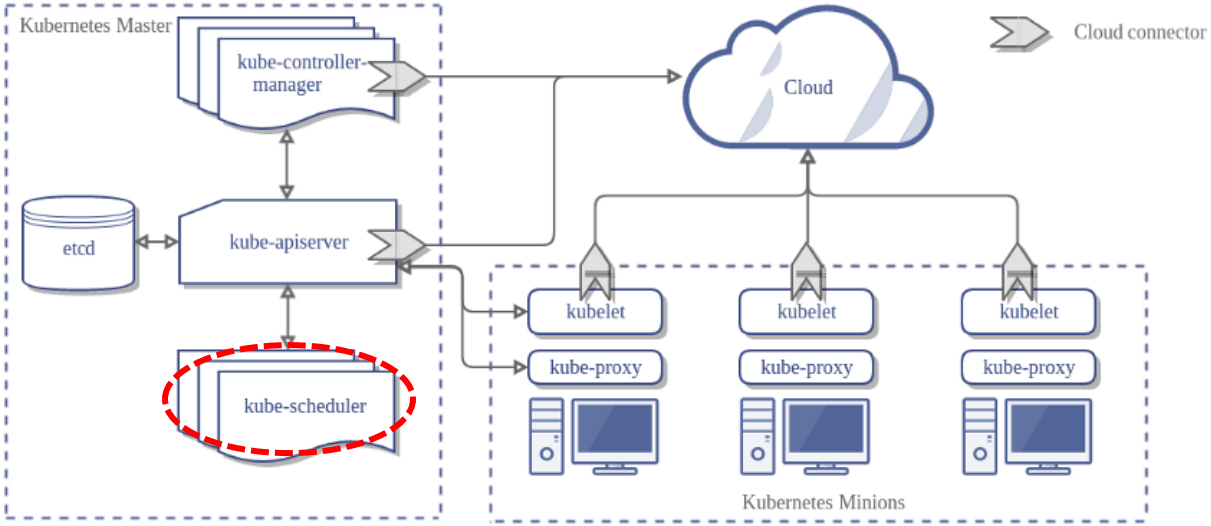
Kube-apiserver

- 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



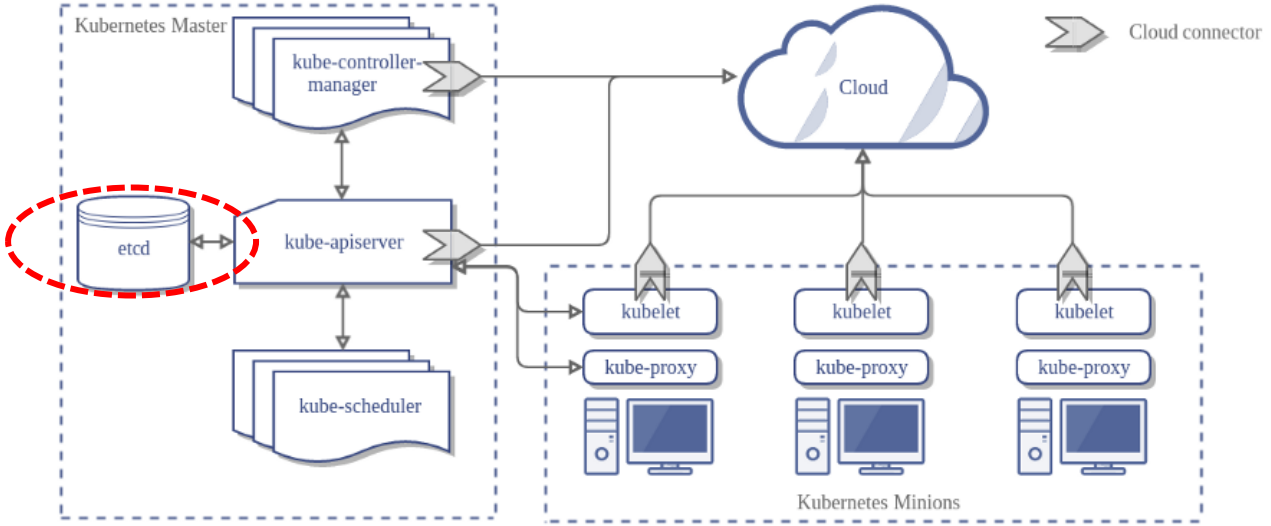
kube-scheduler

- 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



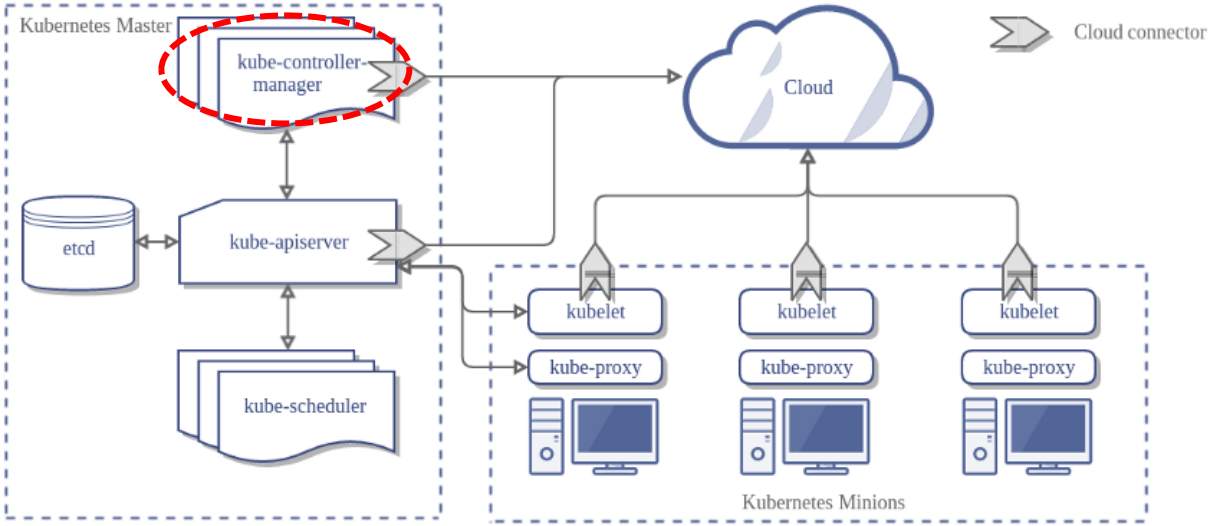
etcd

- 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)



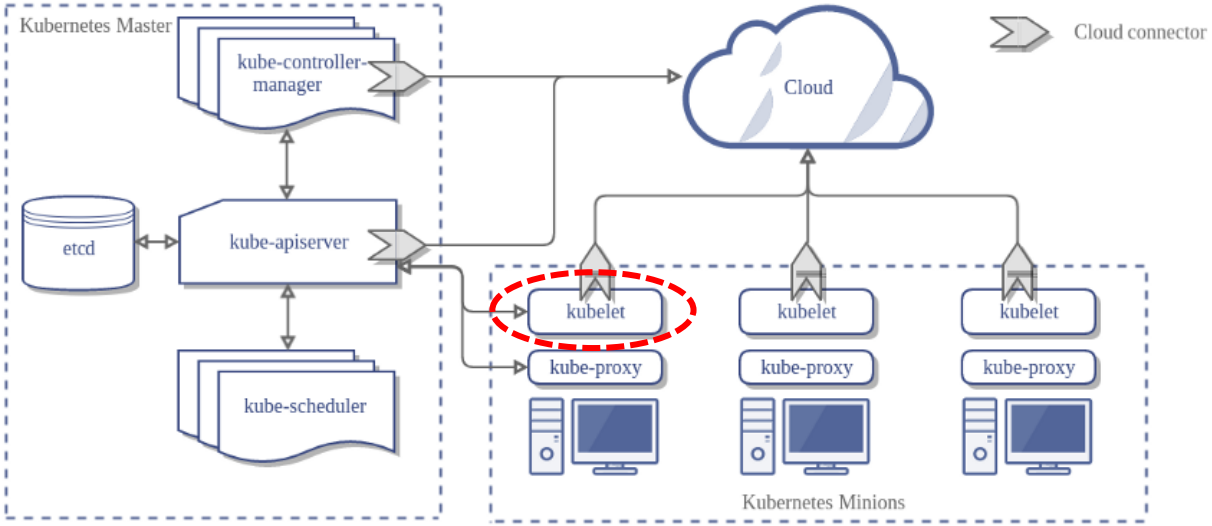
kube-controller-manager

- 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
  - ...



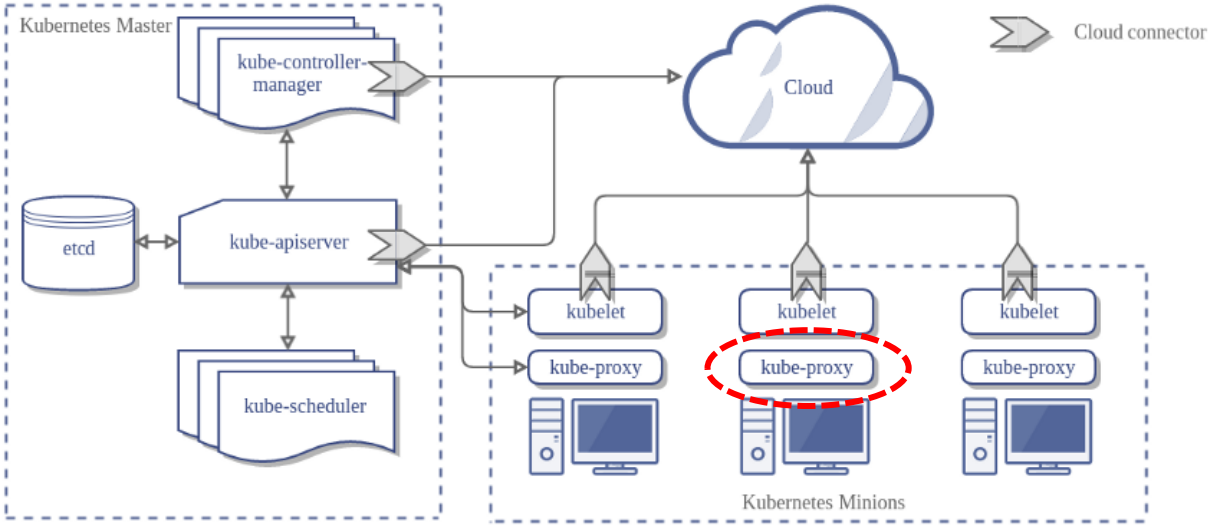
kubelet

- 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**
- 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
- 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



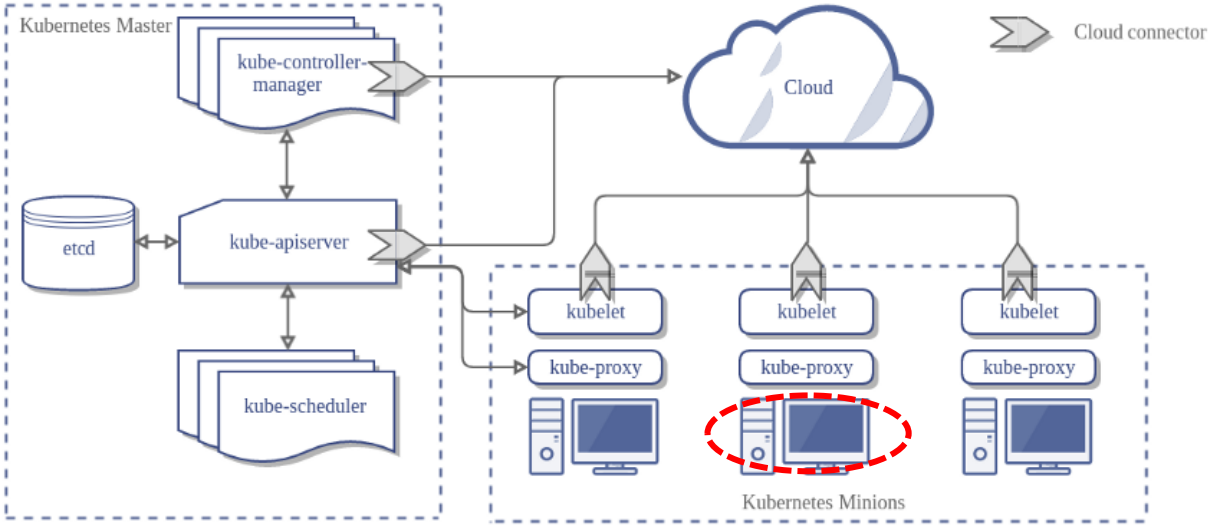
kube-proxy

- 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