容器与云原生

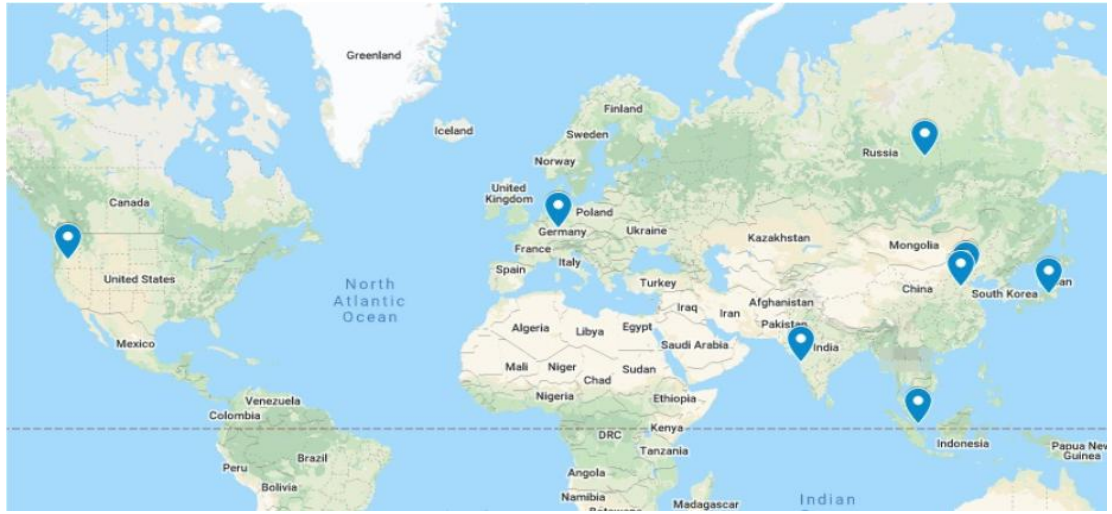
chenshaowen

关于我

- 陈少文
- 金山办公，容器开发工程师
- 研发 PaaS、CICD 平台
- site: www.chenshaowen.com

关于所在团队 - SRE 团队

- 国内（两区）
- 海外（六区）
- 金山云、AWS、华为云、自建机房



金山云

HUAWEI

AWS

关于所在团队 - 超 80%的容器化率

- 金山办公成立于 1988 年
- 2015 年开始落地 Kubernetes
- 超过百个 Kubernetes 集群
- 孵化出的应用管理平台已经运行上数千应用，每天处理数百亿请求

容器的理解与使用

目录

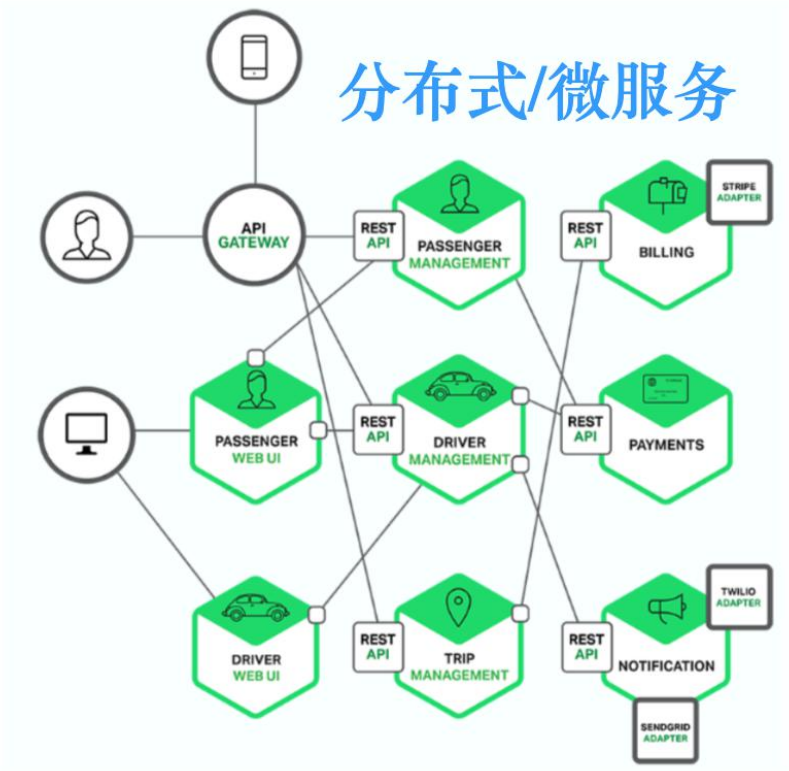
应用数量的急剧增长

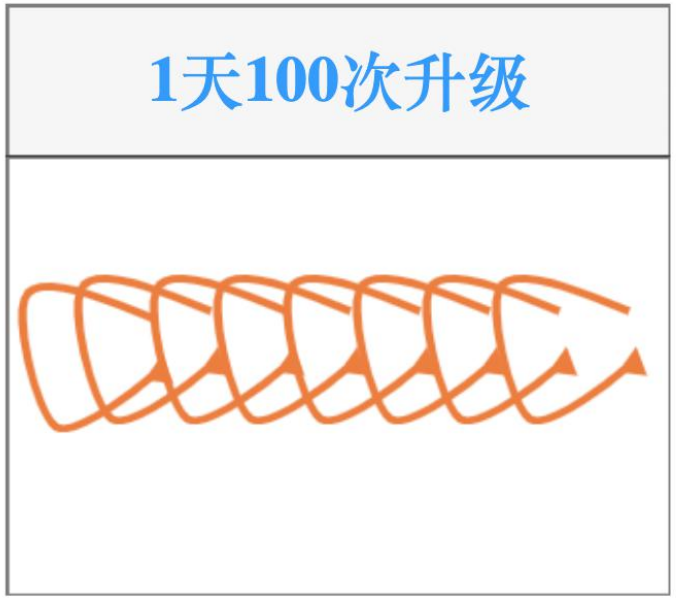
生活场景中，多久没有打开过网站？

2021 年 9 月末，我国国内市场上监测到的APP数量为274万款。

其中，本土第三方应用商店APP数量为138万款，苹果商店（中国区）APP数量为136万款。

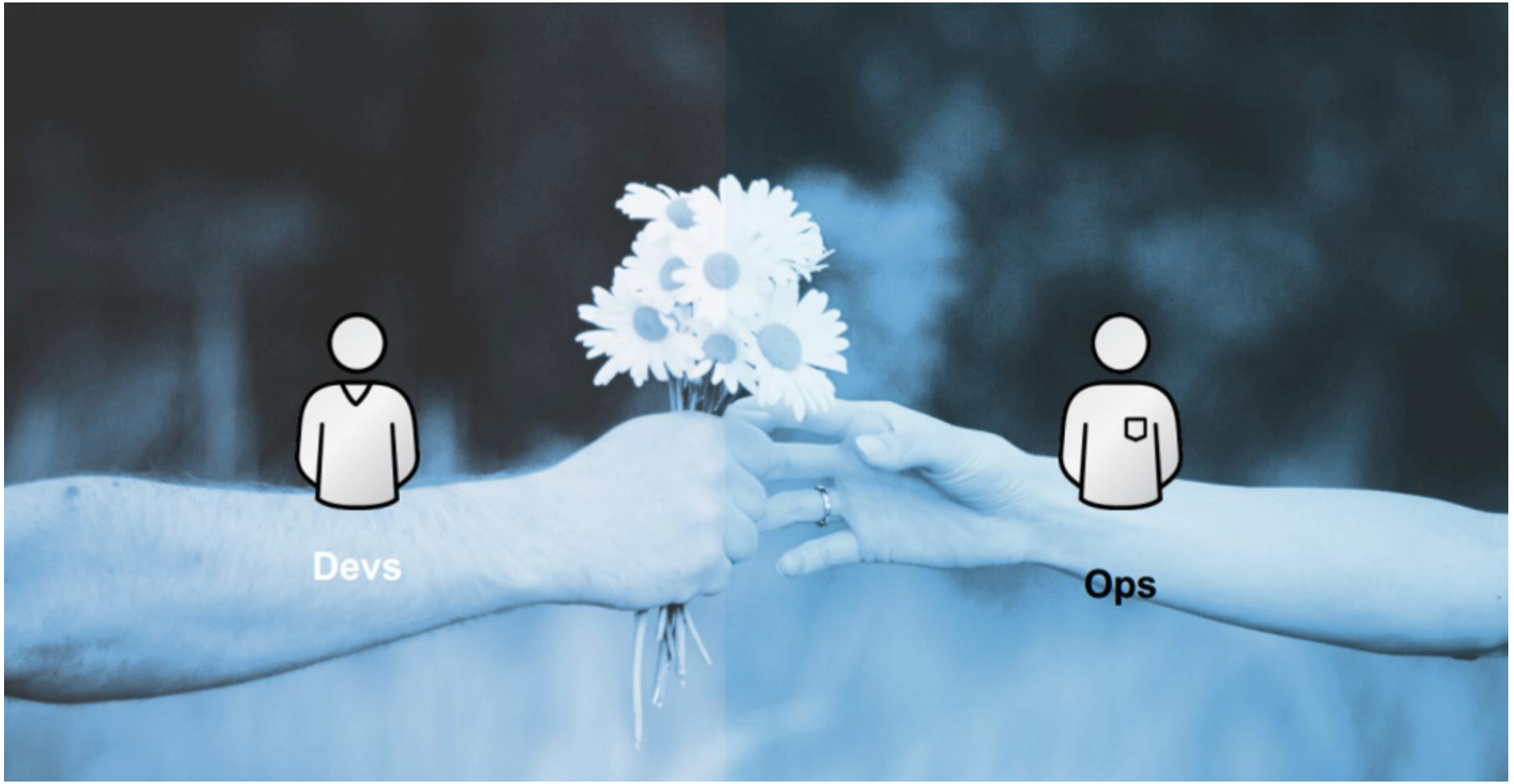
后台服务数量急剧增长，越来越快的更新





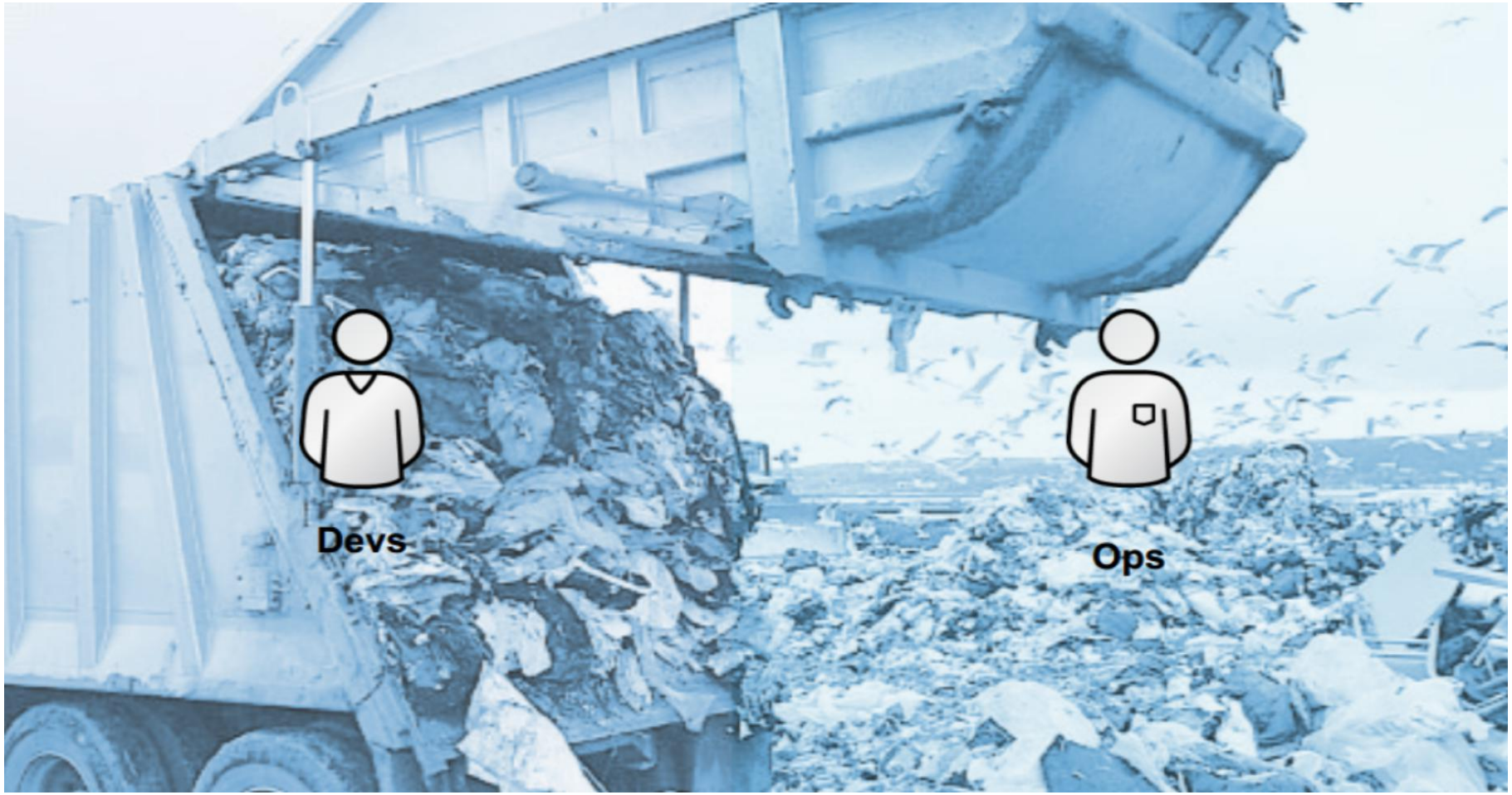
- 应用服务数量增加
- 微服务数量增加
- 发布频率增加
- 运行环境越来越复杂

开发人员认为交付的是...



一个开发的骄傲

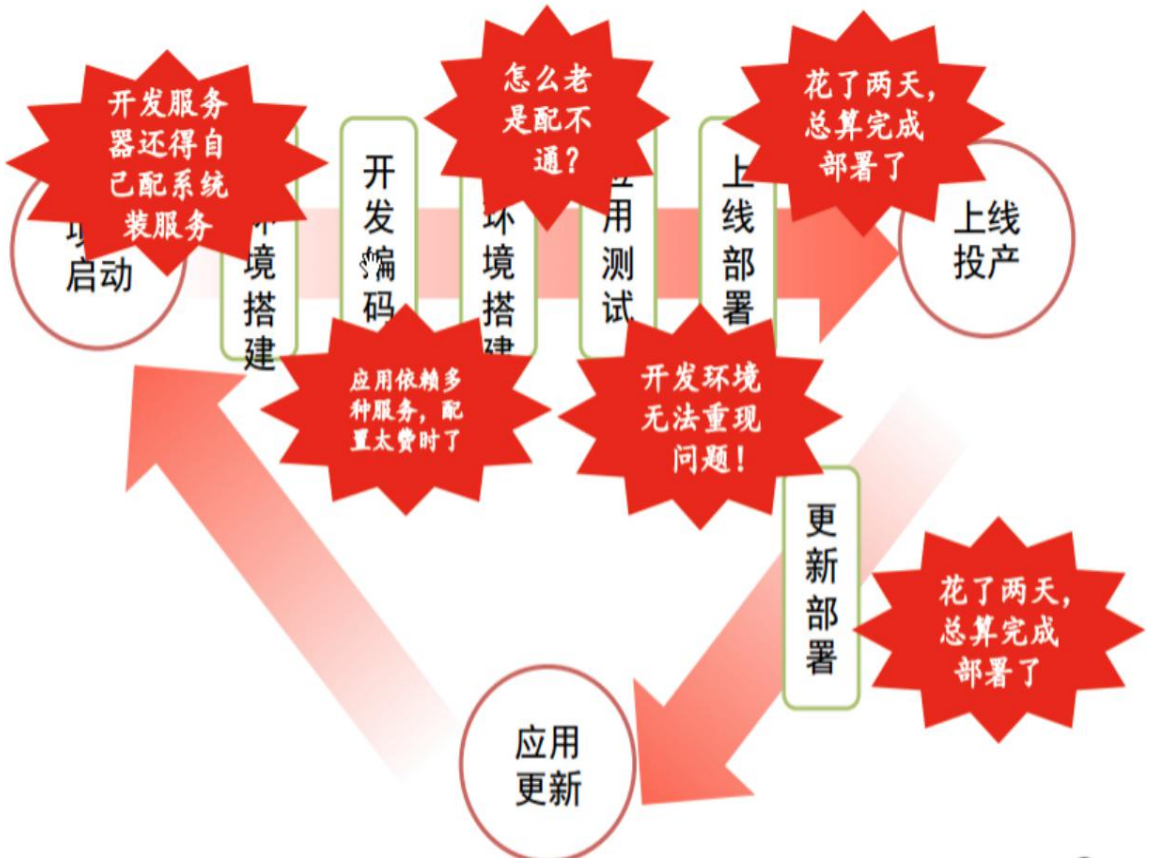
运维人员认为交付的是...



以为是卖家秀
实际是买家秀

你以为的不是
别人以为的

从开发到上线的各种问题



一直开发一直爽

上线时暴露的

问题越来越多

影响效率、制造事故

=> 容器、DevOps

容器技术的发展史

1979 年，Unix v7 系统支持 chroot，为应用构建一个独立的虚拟文件系统视图。

1999 年，FreeBSD 4.0 支持 jail，第一个商用化的 OS 虚拟化技术。

2004 年，Solaris 10 支持 Solaris Zone，第二个商用化的 OS 虚拟化技术。

2005 年，OpenVZ 发布，非常重要的 Linux OS 虚拟化技术先行者。

2004 年 ~ 2007 年，Google 内部大规模使用 Cgroups 等的 OS 虚拟化技术。

2006 年，Google 开源内部使用的 process container 技术，后续更名为 cgroup。

2008 年，Cgroups 进入了 Linux 内核主线。

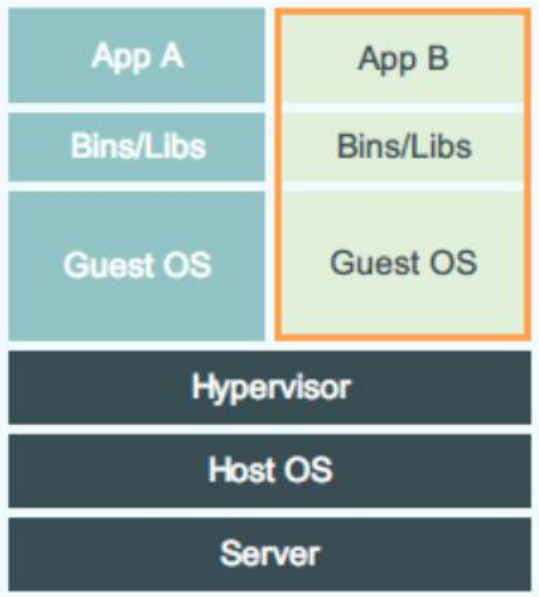
2008 年，LXC（Linux Container）项目具备了 Linux 容器的雏型。

2011 年，CloudFoundry 开发 Warden 系统，一个完整的容器管理系统雏型。

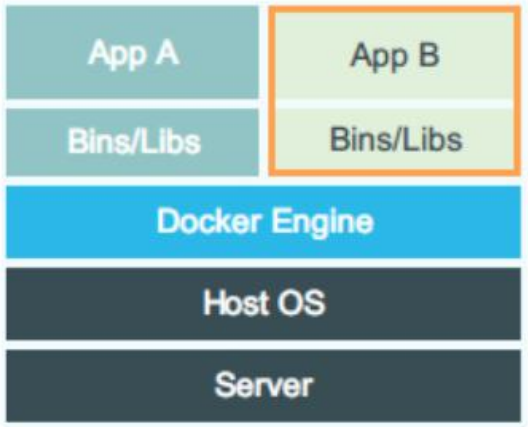
2013 年，Google 通过 Let Me Contain That For You (LMCTFY) 开源内部容器系统。

2013 年，Docker 项目正式发布，让 Linux 容器技术逐步席卷天下

虚拟机 VS 容器



虚拟机



容器

- 虚拟机技术是虚拟一个完整操作系统，在该系统上再运行所需应用进程
- 容器是直接运行于宿主的内核
- Docker 技术在容器的基础上，进行了进一步的封装，从文件系统、网络互联到进程隔离等等，极大的简化了容器的创建和维护，使得Docker技术比虚拟机技术更为轻便、快捷。

容器 VS 虚拟机

特性	容器	虚拟机
启动	秒级	分钟级
硬盘使用	一般为 MB	一般为 GB
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个

容器的特点:

- 轻量级
- 沙箱
- 扩容
- 迁移

容器支持技术

容器的主要由 namespace 和 cgroup 技术支撑

- namespace

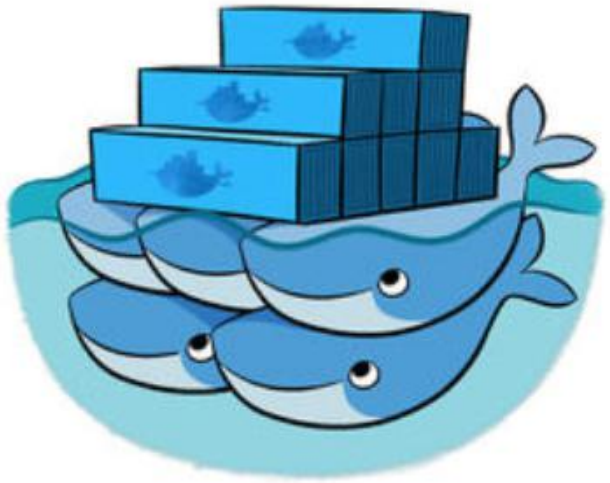
  将一组相关的进程放在一个 ns 中。同一个 ns 中的进程能够相互通信，但是不同 ns 相互隔离。每个 ns 具有独立的资源空间，包括主机名、PID、IPC、文件、网络、用户。

- cgroup

  对进程能使用的物理资源，CPU、内存等资源进行限制。

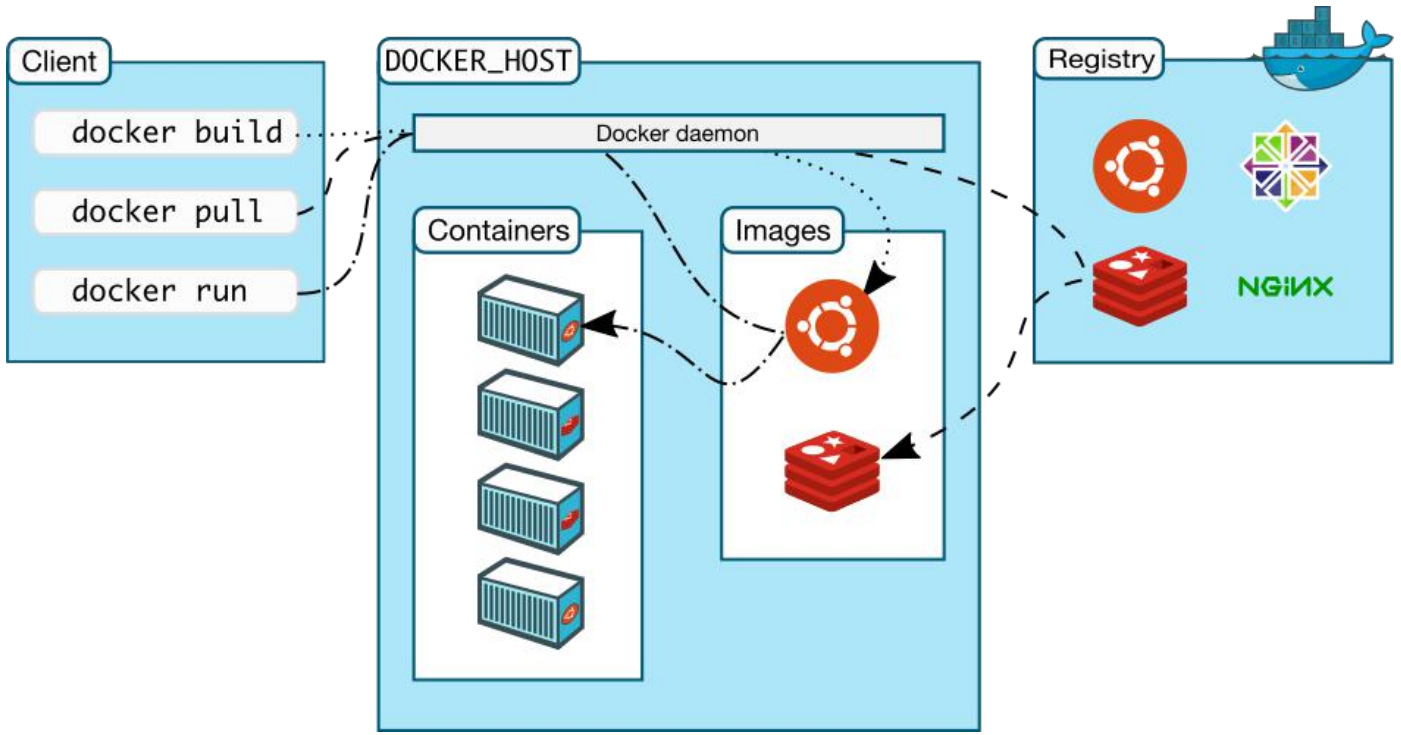
Docker

Docker 是 PaaS 提供商 dotCloud 开源的一个基于 LXC 的高级容器引擎，基于 Go 语言并遵从 Apache2.0 协议开源，托管在 Github 上。



- 打包和发布应用
- 快速部署
- 隔离环境
- 搭建自己的 PaaS 环境

Docker 组件的基本组成



核心组件包括：

客户端 – Docker Client

服务器 - Docker Daemon

仓库 – Registry

Docker 组件的基本组成

- Docker 客户端：

Docker 最常见的客户端就是 docker 命令，通过 docker 命令可以方便地在Host（宿主机）上构建和运行 容器。

- Docker 服务器

Docker deamon 是服务器组件，以 Linux 后台服务的形式运行。默认情况下，docker daemon 只能响应来自本地 Host 的客户端请求。如果要允许远程客户端请求，需要在配置文件中打开 TCP 监听。docker info 可以查看服务器的信息。

随堂实验 01 - 安装 Docker

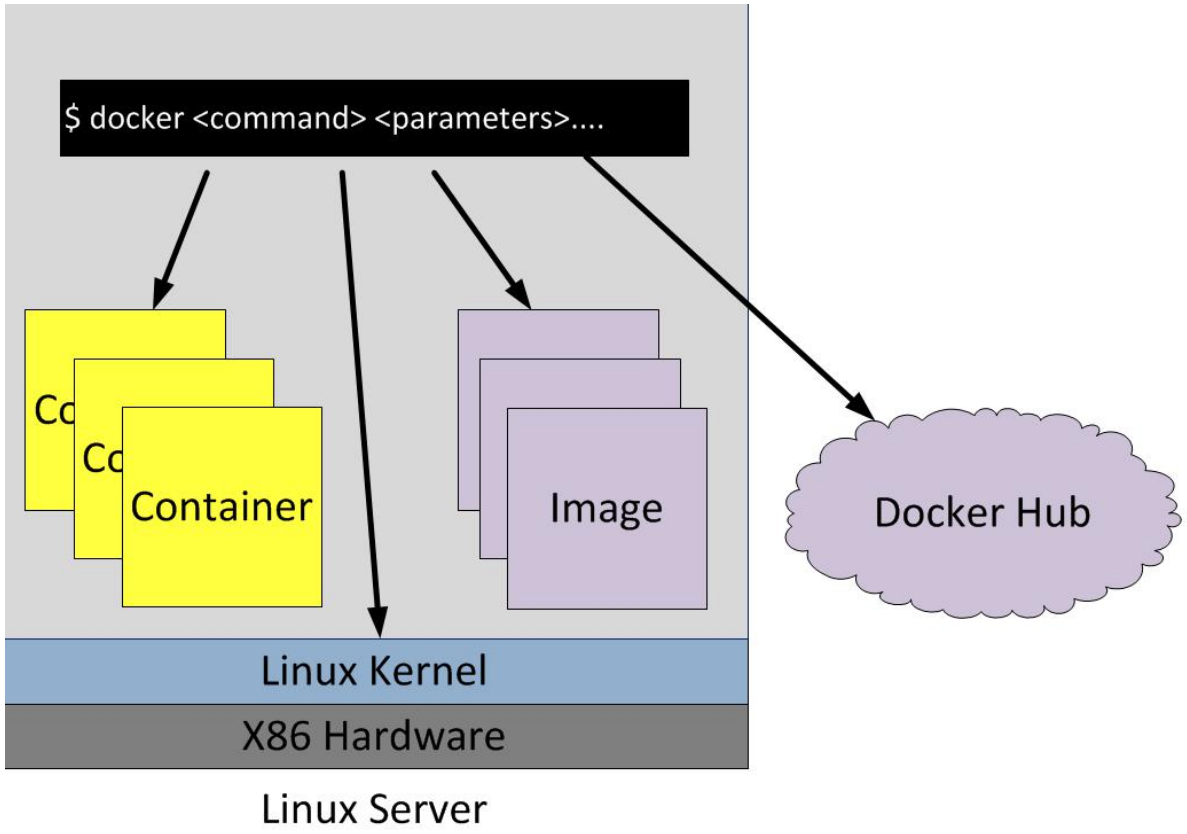
- macOS 安装

  https://docs.docker.com/desktop/mac/install/
- Windows 安装

  https://docs.docker.com/desktop/windows/install/
- Linux 安装

curl -fsSL https://get.docker.com | bash -s docker --mirror Aliyun

Docker 相关命令



- 容器生命周期管理命令

run，start/stop/restart，kill，rm，pause/unpause，create，exec

- 容器操作命令

ps，inspect，top，attach，events，logs，wait，export，port

- 容器rootfs命令

commit，cp，diff

- 镜像仓库命令

login，pull，push，search

- 本地镜像管理命令

images，rmi，tag，build，history，save，import

- info|version命令

info，version

随堂实验 02 - 启动容器

基于镜像新建一个容器并启动

示例1：

docker run busybox /bin/echo 'Hello!'

```
root@node1:~# docker run busybox uname -a
Linux de014263ccc4 5.4.0-81-generic #91-Ubuntu SMP Thu Jul 15 19:09:17 UTC 2021 x86_64 GNU/Linux
root@node1:~# uname -a
Linux node1 5.4.0-81-generic #91-Ubuntu SMP Thu Jul 15 19:09:17 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
```

随堂实验 02 - 启动容器

示例2：

docker run -it centos:7 /bin/bash

启动后，会进入命令行终端交互模式。其中，-t 选项让Docker分配一个伪终端（pseudo-tty）并绑定到容器的标准输入上，-i 则让容器的标准输入保持打开。

随堂实验 02 -启动容器

示例:

```
docker run -id --name mycentos centos:7
docker stop mycentos
docker start mycentos
docker rm mycentos
```

Docker 容器的网络

- 查看默认网络 docker network ls

```
NETWORK ID          NAME                DRIVER              SCOPE
2a194d863ec8        bridge              bridge              local
14934a7190f7        host                host                local
f0cb22723103        none                null                local
```

- 可以通过命令行指定网络类型

host模式：使用 --net=host 指定。
none模式：使用 --net=none 指定。
bridge模式：使用 --net=bridge 指定，默认设置。
container模式：使用 --net=container:NAME_or_ID 指定。

Docker 容器的网络

- Bridge 模式

Docker 在安装时会创建一个名为 docker0 的 Linux 网桥。

bridge 模式是 Docker 默认的网络模式，在不指定 --network 的情况下，Docker 会为每一个容器分配 network namespace、设置 IP 等，并将 Docker 容器连接到 docker0 网桥上。

Docker 容器的网络

- Host 模式

连接到 host 网络的容器共享 Docker host 的网络栈，容器的网络配置与 host 完全一样。host 模式下容器将不会获得独立的 network namespace，而是和宿主机共用一个network namespace。容器将不会虚拟出自己的网卡，配置自己的IP等，而是使用宿主机的IP和端口。

Docker 容器的网络

- Container 模式

container 模式指定新创建的容器和已经存在的任意一个容器共享一个 network namespace，但不能和宿主机共享。新创建的容器不会创建自己的网卡，配置自己的IP，而是和一个指定的容器共享IP、端口范围等。同样，两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的。

Kubernetes 中的 Pod 采用的就是这种方式。

问答01

容器能跑在哪些环境上?

A、x86物理机
B、虚拟机
C、arm 物理机
D、云主机

问答02

为什么容器比虚拟机资源效率更高？

问答03

Docker 容器有哪些常用的命令?

随堂实验 03 - 容器的基本操作

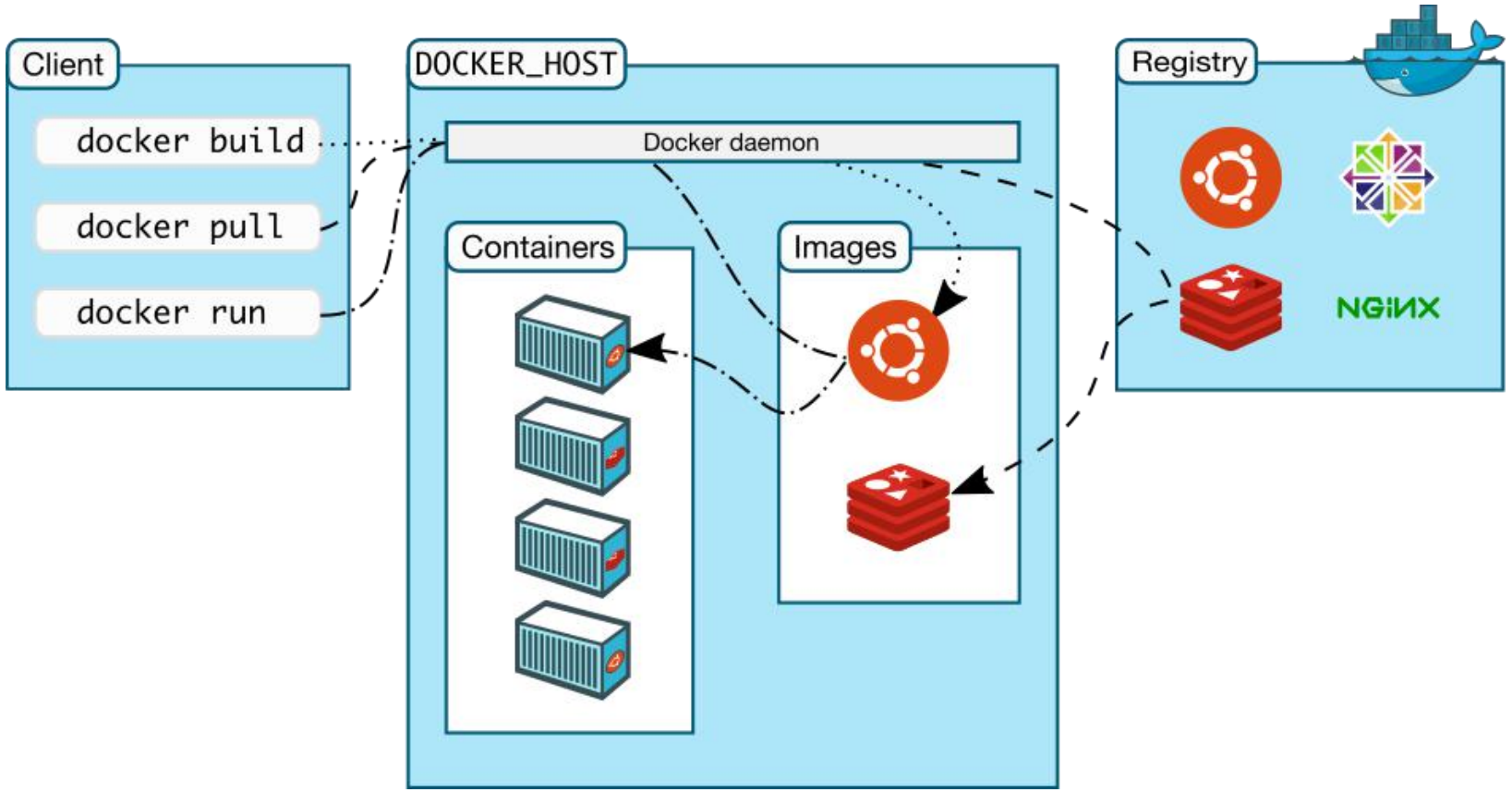
运行一个 Nginx 容器，并通过本地 8081 端口访问首页。

- 运行一个容器
- 查看容器在宿主机上的日志文件
- 查看容器的日志
- 怎么将本地目录挂载到容器中？volume
- 停止容器

容器镜像

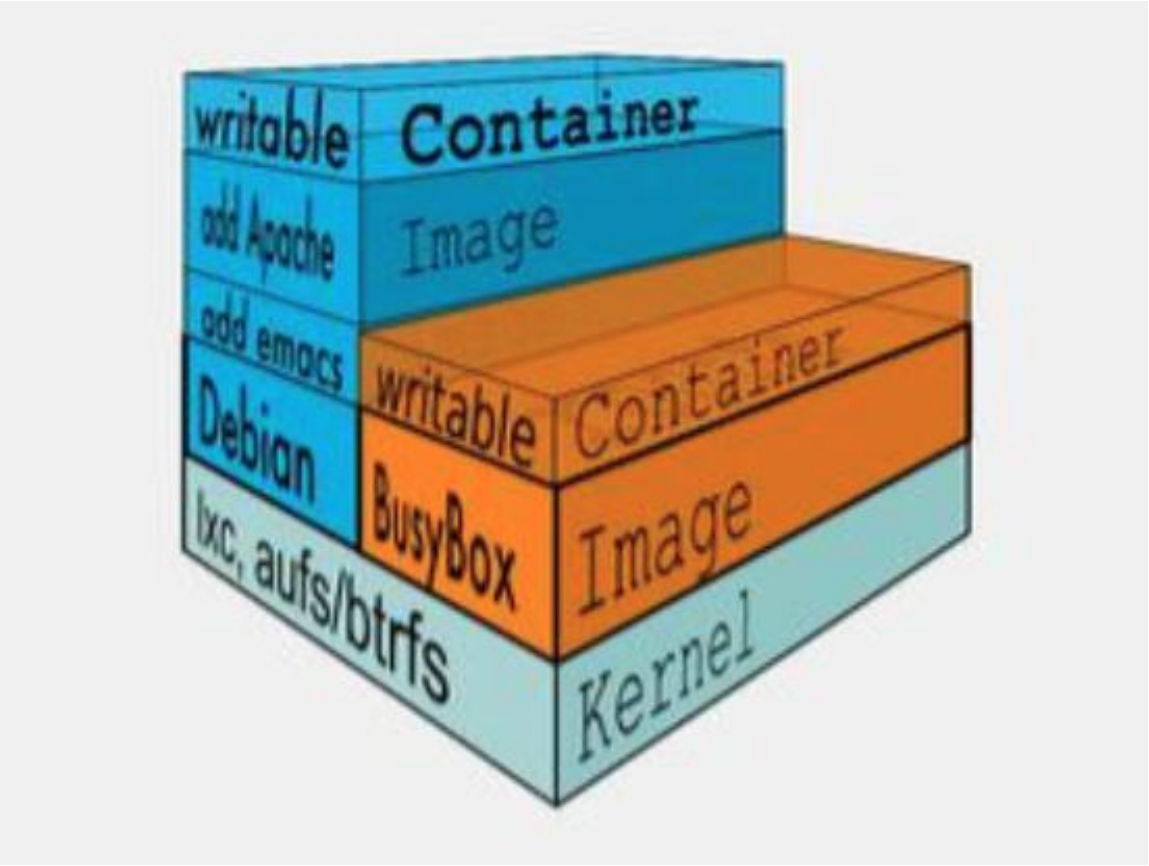
目录

容器镜像



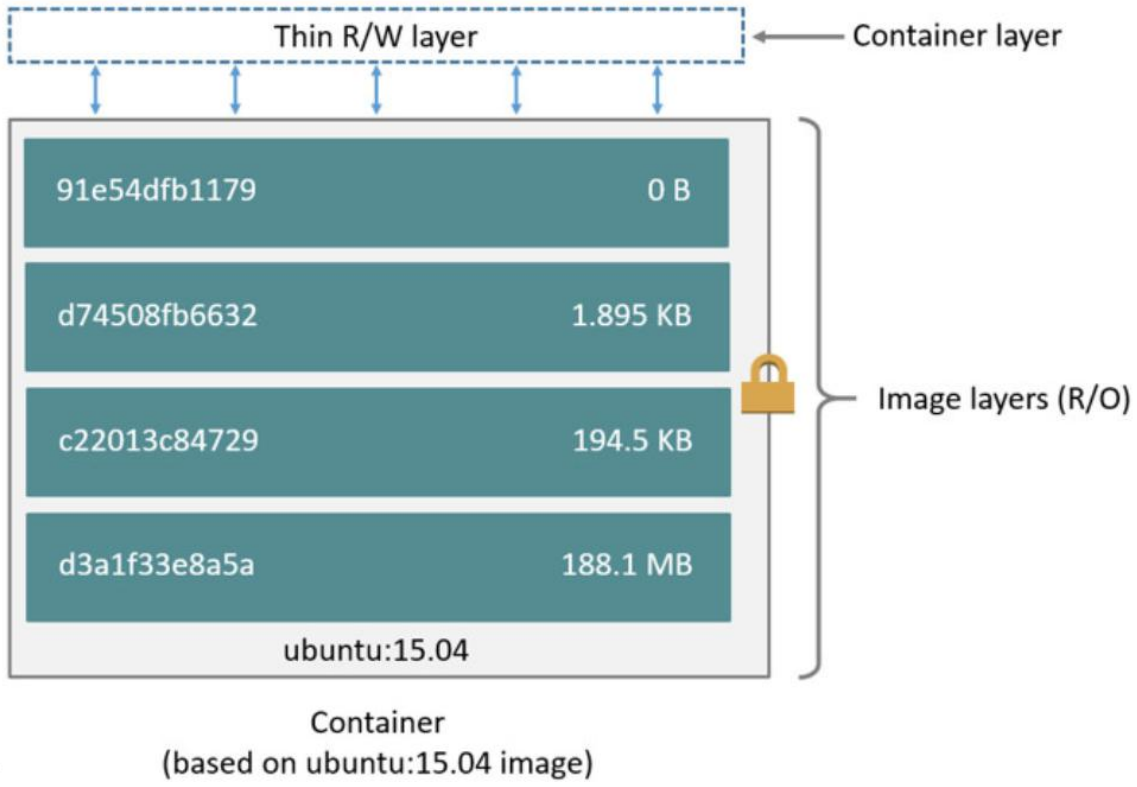
- 镜像存储于远程服务端
- 使用时，被拉取到本地

容器镜像



- 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。
- 镜像不包含任何动态数据，其内容在构建之后也不会被改变。

镜像的分层存储



- 镜像并非是像一个 ISO 那样的打包文件，镜像只是一个虚拟的概念，其实际体现并非由一个文件组成，而是由一组文件系统组成，或者说，由多层文件系统联合组成
- 镜像构建时分层构建，前一层是后一层的基础。每一层构建完就不会再发生改变，后一层上的任何改变只发生在自己这一层
- 分层存储有利于镜像的复用、定制

镜像的制作 Dockerfile

Dockerfile 示例：

```
FROM alpine
COPY ./bin/kaeappdeploy ./bin/
RUN chmod +x ./bin/kaeappdeploy
CMD ["./bin/kaeappdeploy"]
```

编译命令：docker build -t shaowenchen/test:v1 .

Dockerfile 常用命令

- FROM

基于哪个镜像，其中 scratch 是空镜像

- RUN

在基础镜像上，执行命令，然后提交到新的镜像上

- COPY

将源文件拷贝到镜像中

- CMD

启动容器时，运行的命令

随堂实验 01 - Docker 镜像制作过程

制作流程：

1. 写 Dockerfile

2. docker build -t 镜像名字 -f Dockerfile .

镜像制作 - Python

源码:
https://github.com/prakhar1989/docker-curriculum/blob/master/flask-app

Dockerfile：

```
FROM python:3

WORKDIR /usr/src/app

COPY . .

RUN pip install --no-cache-dir -r requirements.txt

EXPOSE 5000

CMD ["python", "./app.py"]
```

构建：docker build -t flask-test:0.1

运行：docker run -d -p 8080:5000 --name myflask flask-test:0.1

镜像制作 - Java

源码:

https://github.com/xuxueli/xxl-job.git

本地构建:

cd xxl-job && mvn clean package

Dockerfile:

```
FROM openjdk:8-jre-alpine
COPY xxl-job-admin/target/xxl-job-admin-2.1.0-SNAPSHOT.jar /srv
EXPOSE 8080
CMD ["/usr/bin/java", "-jar", "/srv/xxl-job-admin-2.1.0-SNAPSHOT.jar"]
```

镜像仓库

- 仓库是集中存放Docker镜像文件的场所，在Docker本地运行环境中直接从仓库拉一个镜像到本地来使用。
- 仓库有公共仓库和私有仓库

  最大的公共仓库就是 Docker Hub，国内也有网易云等提供Docker仓库。私有仓库是企业内部建立和使用的。

常见的镜像仓库

- Registry

  docker run -d -v /edc/images/registry:/var/lib/registry -p 5000:5000 registry
- Docker.io

  匿名用户，每 6 小时只允许 pull 100 次

  已登录用户，每 6 小时只允许 pull 200 次
- Harbor

  私有仓库，生产使用
- GCR.IO
- GHCR.IO

如何获取镜像

当 pull 一个镜像时，先进行认证获取到 token 并授权通过

- 获取镜像的 manifest 文件，进行 signature 校验。
- 根据 manifest 里的层信息并发拉取各层

其中 manifest 包含的信息有：仓库名称、tag、镜像层 digest 等

Docker 如何区分多个仓库

docker pull [选项] [Docker Registry 地址[:端口号]/]仓库名[:标签]

通过前缀，区分不同的镜像仓库源

- nginx:latest

向 docker.io（默认） 请求 nginx:latest 镜像数据

- myregistry.com:5000/nginx:latest

向 myregistry.com:5000 请求 nginx:latest 镜像数据。类似的有：

mytest.com:5000/nginx，myprod.com/nginx:latest

镜像的基本操作 - 列出镜像

docker image ls 或 docker images

```
→  ~ docker images
REPOSITORY   TAG              IMAGE ID       CREATED        SIZE
redis        latest           40c68ed3a4d2   4 days ago     113MB
nginx        1-perl           798c4c8fb434   5 days ago     188MB
nginx        1                ea335eea17ab   5 days ago     141MB
nginx        latest           ea335eea17ab   5 days ago     141MB
debian       latest           827e5611389a   5 days ago     124MB
nginx        1-alpine-perl    4d1246c808b0   9 days ago     57.9MB
nginx        1-alpine         b46db85084b8   9 days ago     23.2MB
golang       1-alpine3.13     a9baab48e7f2   9 days ago     315MB
golang       1-alpine         3a38ce03c951   9 days ago     315MB
golang       1-alpine3.14     3a38ce03c951   9 days ago     315MB
alpine       3.11             a787cb986503   9 days ago     5.62MB
alpine       3.11.13          a787cb986503   9 days ago     5.62MB
alpine       3.12             b0925e081921   9 days ago     5.59MB
alpine       3.12.9           b0925e081921   9 days ago     5.59MB
alpine       3.13             6b7b3256dabe   9 days ago     5.62MB
alpine       3.13.7           6b7b3256dabe   9 days ago     5.62MB
alpine       3                0a97eee8041e   9 days ago     5.61MB
alpine       3.14             0a97eee8041e   9 days ago     5.61MB
alpine       3.14.3           0a97eee8041e   9 days ago     5.61MB
alpine       latest           0a97eee8041e   9 days ago     5.61MB
busybox      latest           7138284460ff   10 days ago    1.24MB
golang       1-buster         848fd7d0fb9d   2 weeks ago    884MB
```

镜像的基本操作 - 删除镜像

docker image rm [选项] <镜像1> [<镜像2> ...]

简写形式：

docker rmi [选项] <镜像1> [<镜像2> ...]

```
→  ~ docker rmi 2f3c6710d8f2
Untagged: nginx:1.11.5-alpine
Untagged: nginx@sha256:5c6be3514a00db611371806c5aeb510dfd285a4b8261c6e5c3323387e396b66b
Deleted: sha256:2f3c6710d8f2091223a9e43d2dc306ab1d6064faf90a482041cbf900fe0fdf15
Deleted: sha256:7cd3d9301c662554430fd291db3fa832df9c7df1c0c5c39467647f967476f4ee
Deleted: sha256:243c8853bb9e0faac7be670aa7de30e4611119da356405637d88a40b9ea9b496
Deleted: sha256:0eafa965561f0c086afcdae97813d38093f910ed2dab86146b9a9e18a6af906e
```

镜像的基本操作 - 重命名镜像

docker tag <原镜像名:原tag> <新镜像名:新tag>

注意：重新打 tag 不会删除原镜像。新镜像与原镜像的 Image ID相同。

Docker 镜像仓库命令

搜索指定仓库内的镜像：

docker search 仓库地址/镜像名

登录仓库：

docker login 仓库地址

拉取仓库内的镜像：

docker pull 仓库地址/镜像名

推送本地镜像至仓库：

docker tag 本地镜像 远端仓库地址/镜像名

默认仓库地址是：

docker push

使用镜像仓库常见问题

- 镜像仓库证书错误

  在 /etc/docker/daemon.json 文件中，增加非安全仓库地址：

  ```
  {      "insecure-registries": ["10.10.10.10:5000"]   }
  ```

- 修改镜像源，加速镜像拉取

  在 /etc/docker/daemon.json 文件中，增加镜像源：

  ```
  {      "registry-mirrors": ["https://docker.mirrors.ustc.edu.cn"]   }
  ```

问答题 01

- 常见的镜像仓库有哪些?
- 镜像分层的好处是什么?

随堂实验 02 - 推送镜像到 Dockerhub

1. 搜索并拉取最新的 Nginx 镜像

2. 将其重命名为 {YOURNAME}/newnginx:v1

3. 推送到 Dockerhub 镜像仓库

云原生时代

目录

PIVOTAL

Pivotal 公司的 Matt Stine 在 2013年首次提出云原生（CloudNative）的概念

2015年，12-Factor、面向微服务、抗脆弱

2017年，可观测性、模块化、可替代性、可处理性

2019年，DevOps、CI/CD、微服务、容器

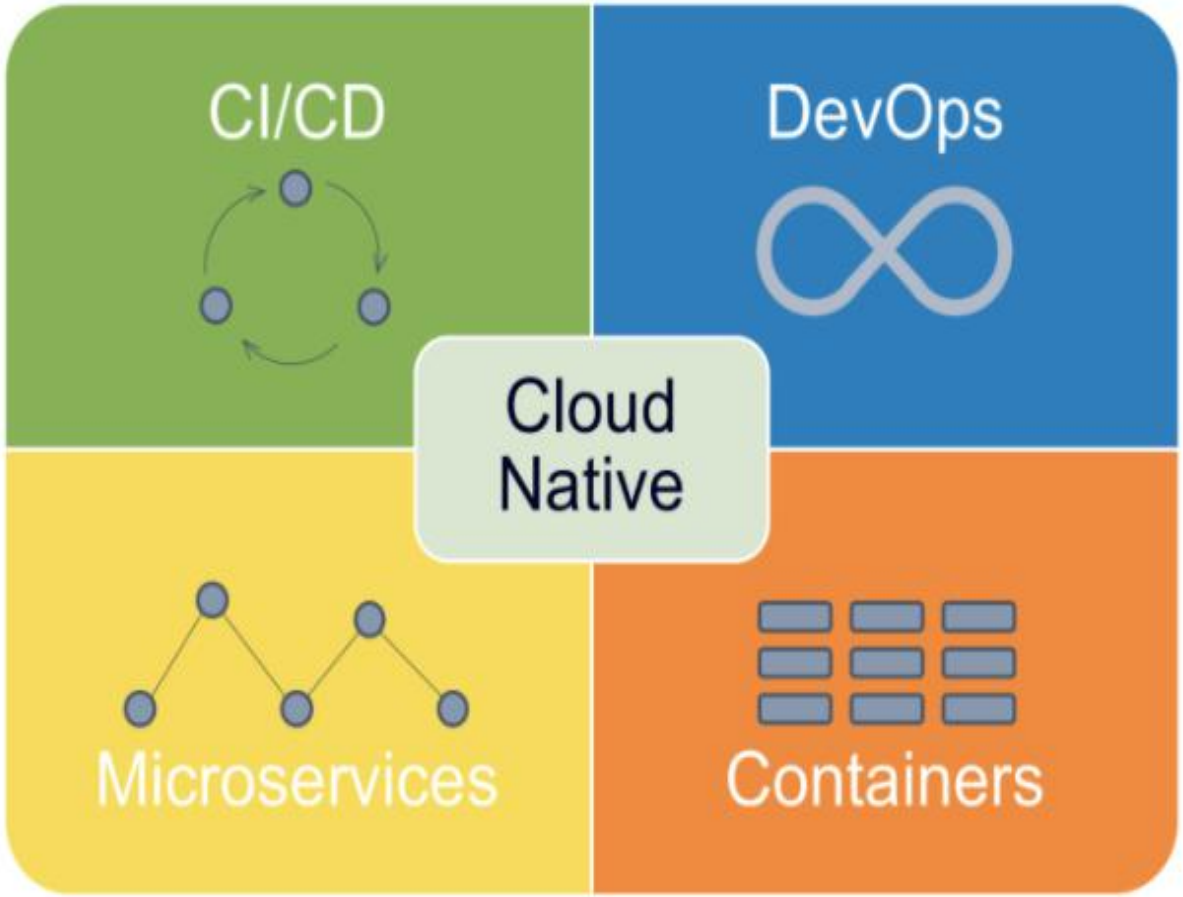
CNCF

CNCF，全称Cloud Native Computing Foundation（云原生计算基金会）

2015年，容器化、微服务、编排调度

2018年，不可变基础设施，声明式 API，服务网格

云原生包含哪些内容

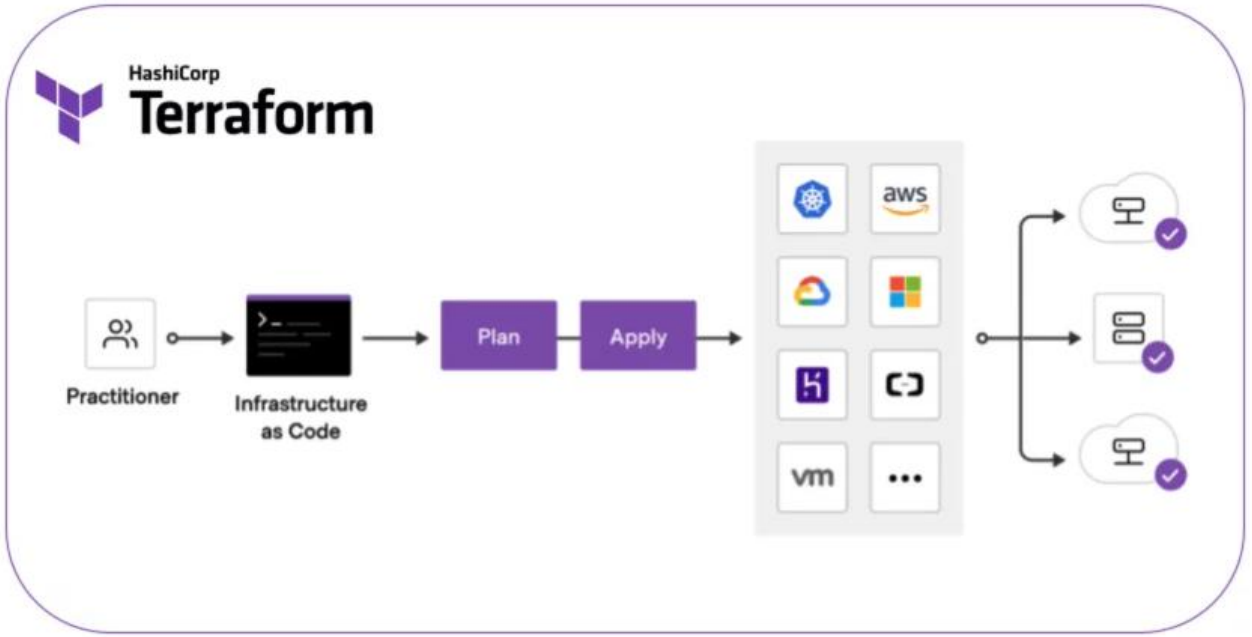


- CICD
- DevOps
- 微服务
- 容器

云原生 - 不可变的基础设施

- 由 Chad Fowler 2013年提出
- 任何基础设施创建后变成为只读状态，只允许替换，不允许修改
  - 环境不一致
  - 配置漂移
  - 禁止 SSH
- 容器能很好地满足要求
- 但运行容器的主机和环境呢？IaC

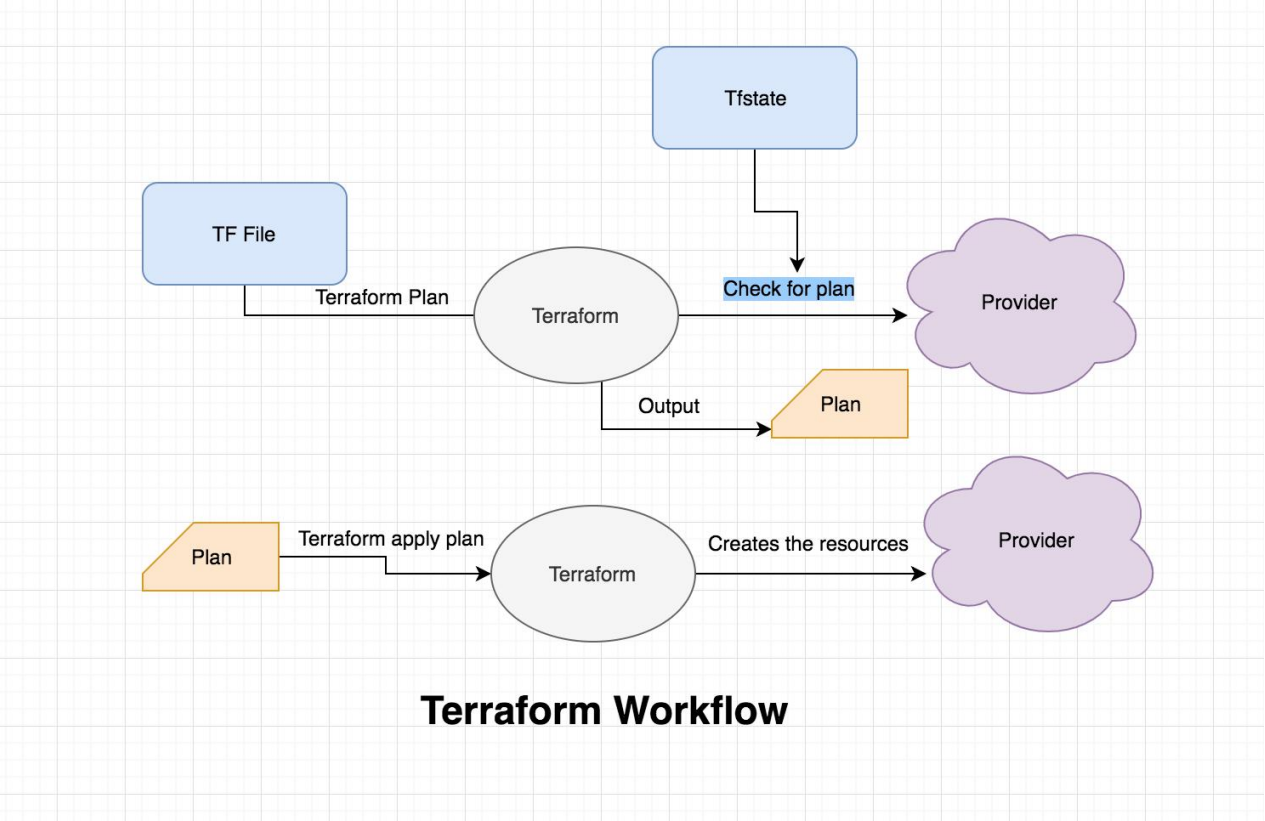
云原生 - IaC



Terraform 是由 HashiCorp 开源的基础架构即代码工具。

Terraform 是一种**声明式编码**工具。可以让开发人员用 HCL（HashiCorp 配置语言）高级配置语言来描述用于运行应用程序的“最终状态”云或本地基础架构。 它随后会生成用于达到该最终状态的计划，并执行该计划以供应基础架构。

云原生 - IaC

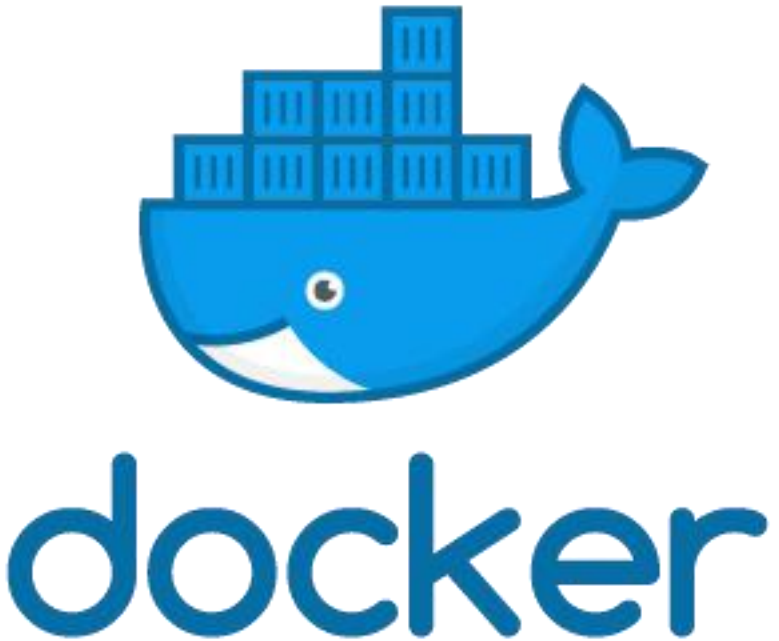



```
resource "qingcloud_instance" "init"{
  count = 1
  name = "master-${count.index}"
  image_id = "centos7x64d"
  instance_class = "0"
  managed_vxnet_id="vxnet-0"
  keypair_ids = ["${qingcloud_keypair.arthur.id}"]
  security_group_id
="${qingcloud_security_group.basic.id}"
  eip_id = "${qingcloud_eip.init.id}"
}
```

terraform apply 一键创建云资源

容器的优点

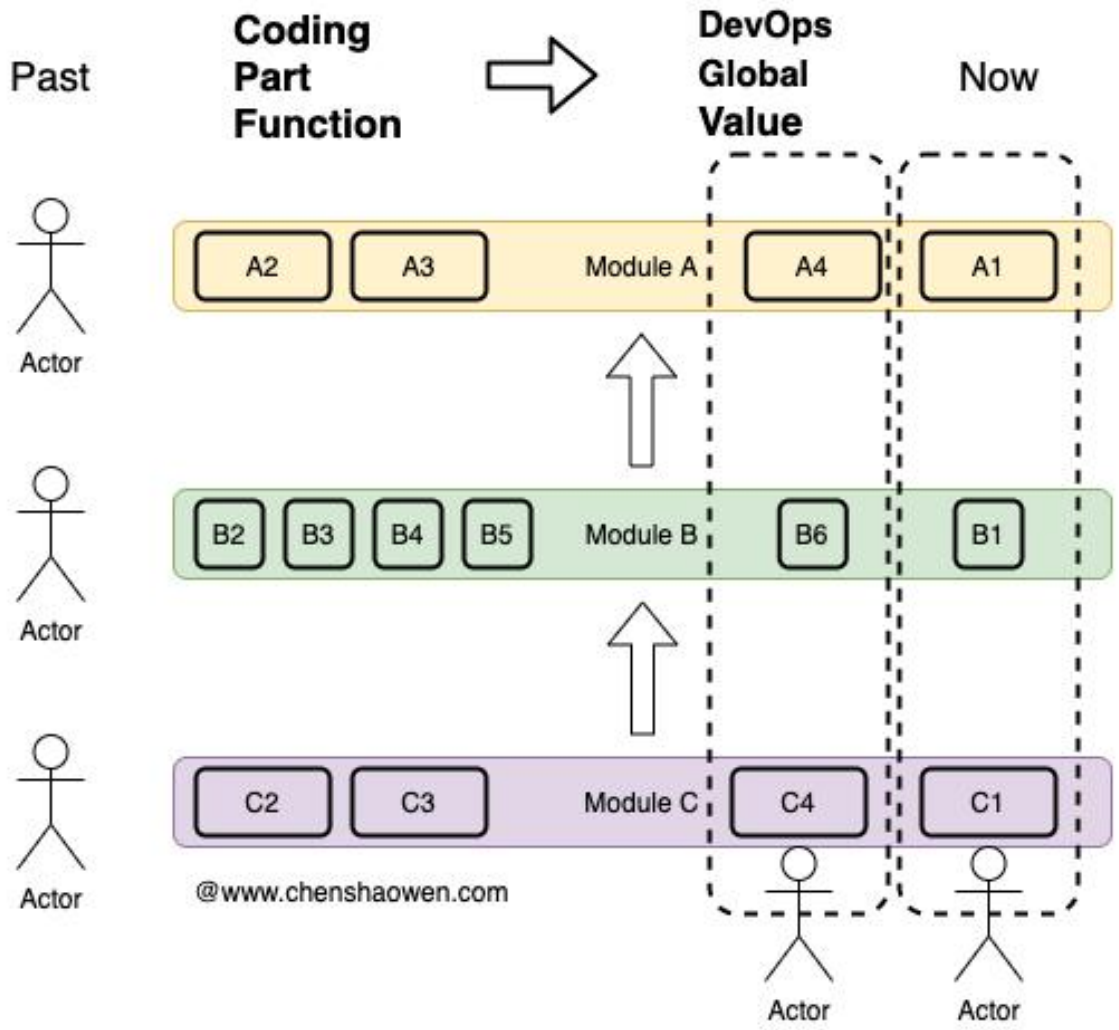
- 更高效的利用系统资源
- 更快速的启动时间
- 一致的运行环境
- 持续交付和部署
- 更轻松的迁移
- 更轻松的维护和扩展



容器的劣势

- 学习成本高
- 爆炸半径大
- 编排复杂
- 隔离性不够

DevOps - 端到端的价值交付



一千个人眼里有一千种 DevOps

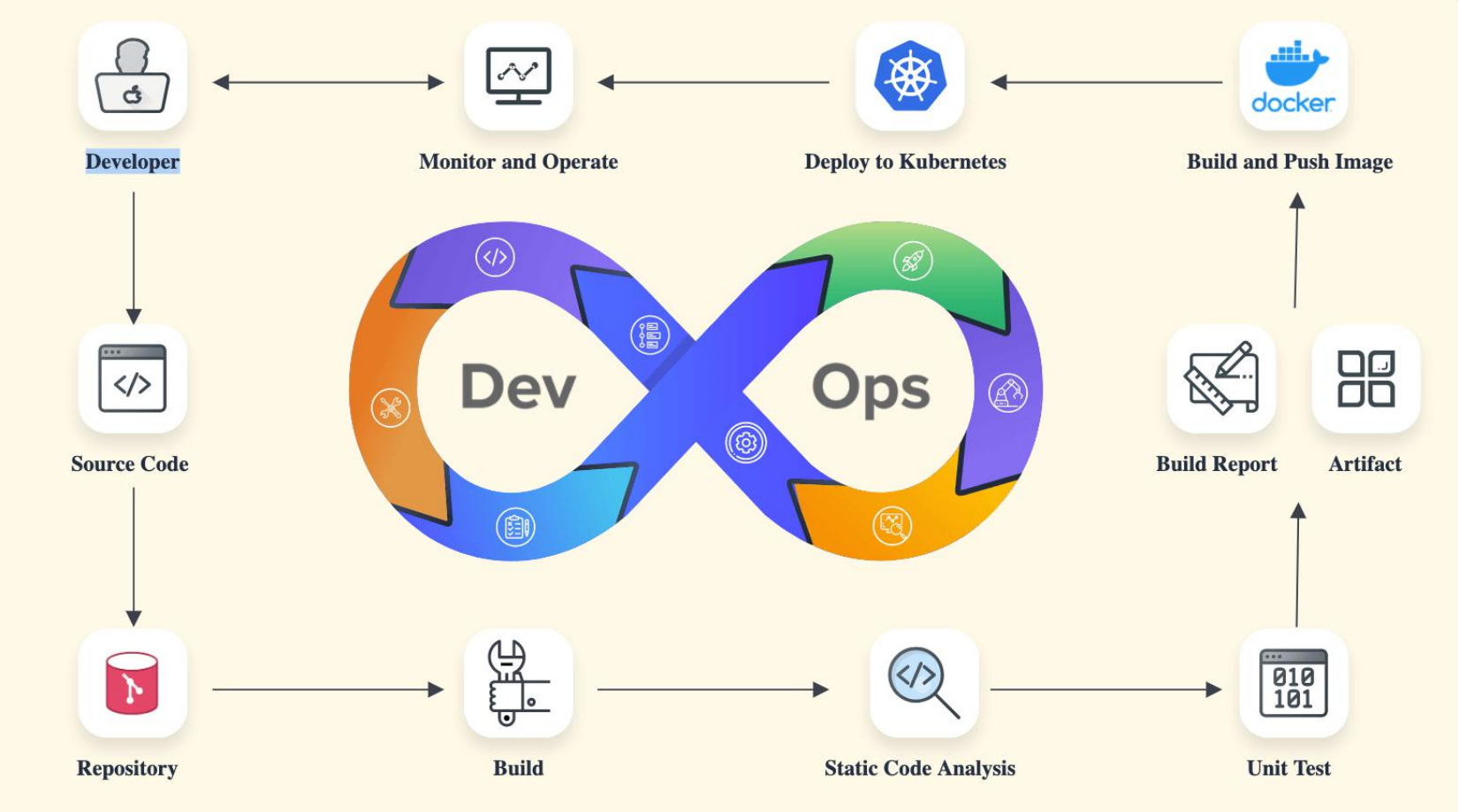
DevOps 是一种文化

DevOps 也是一种工作方法

DevOps 强调的是交付价值

DevOps 需要自动化工具的支撑

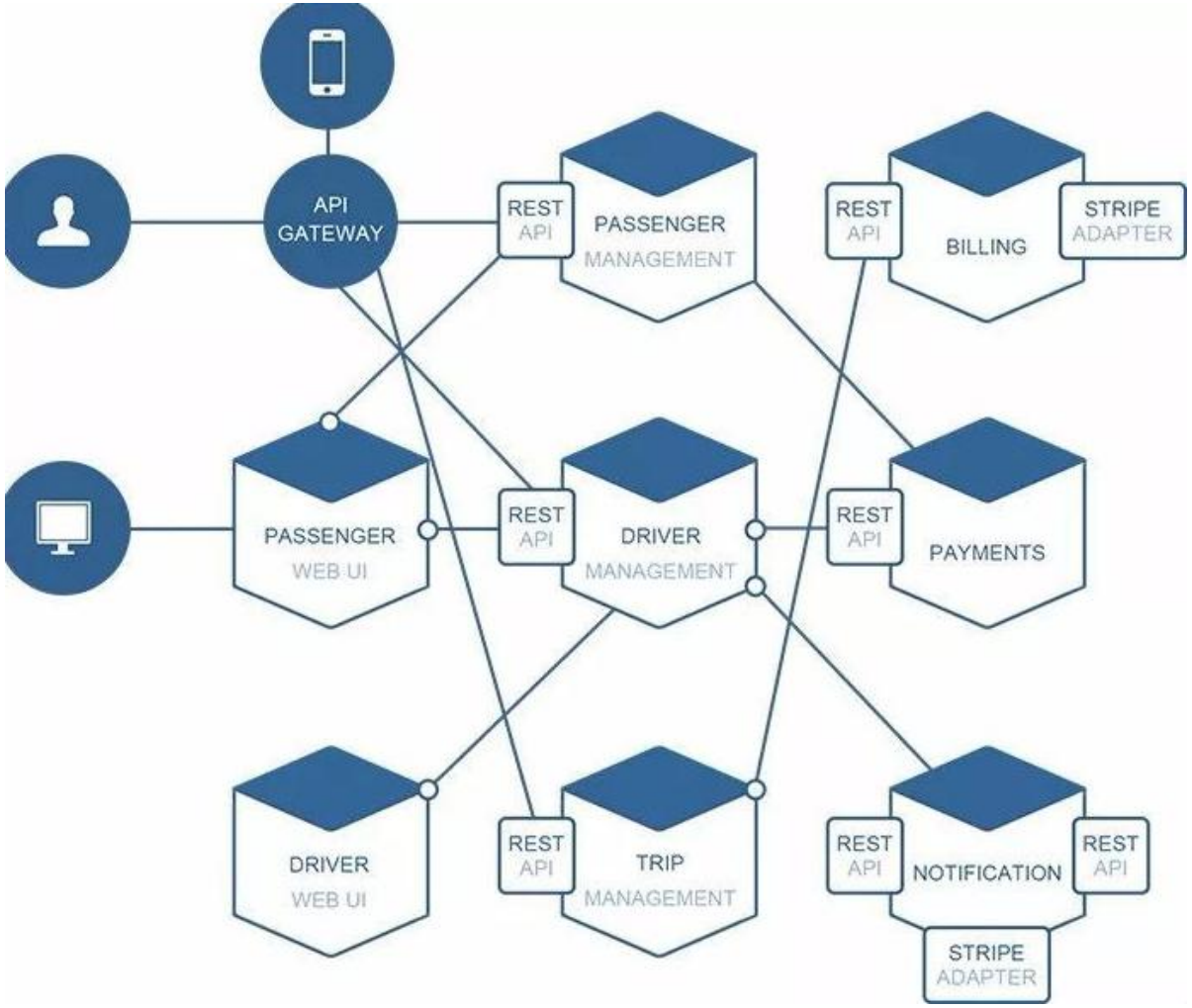
自动化工具



- 需求管理
- 编写代码
- 单元测试
- 进行构建
- 集成测试
- 更新上线
- 完成交付

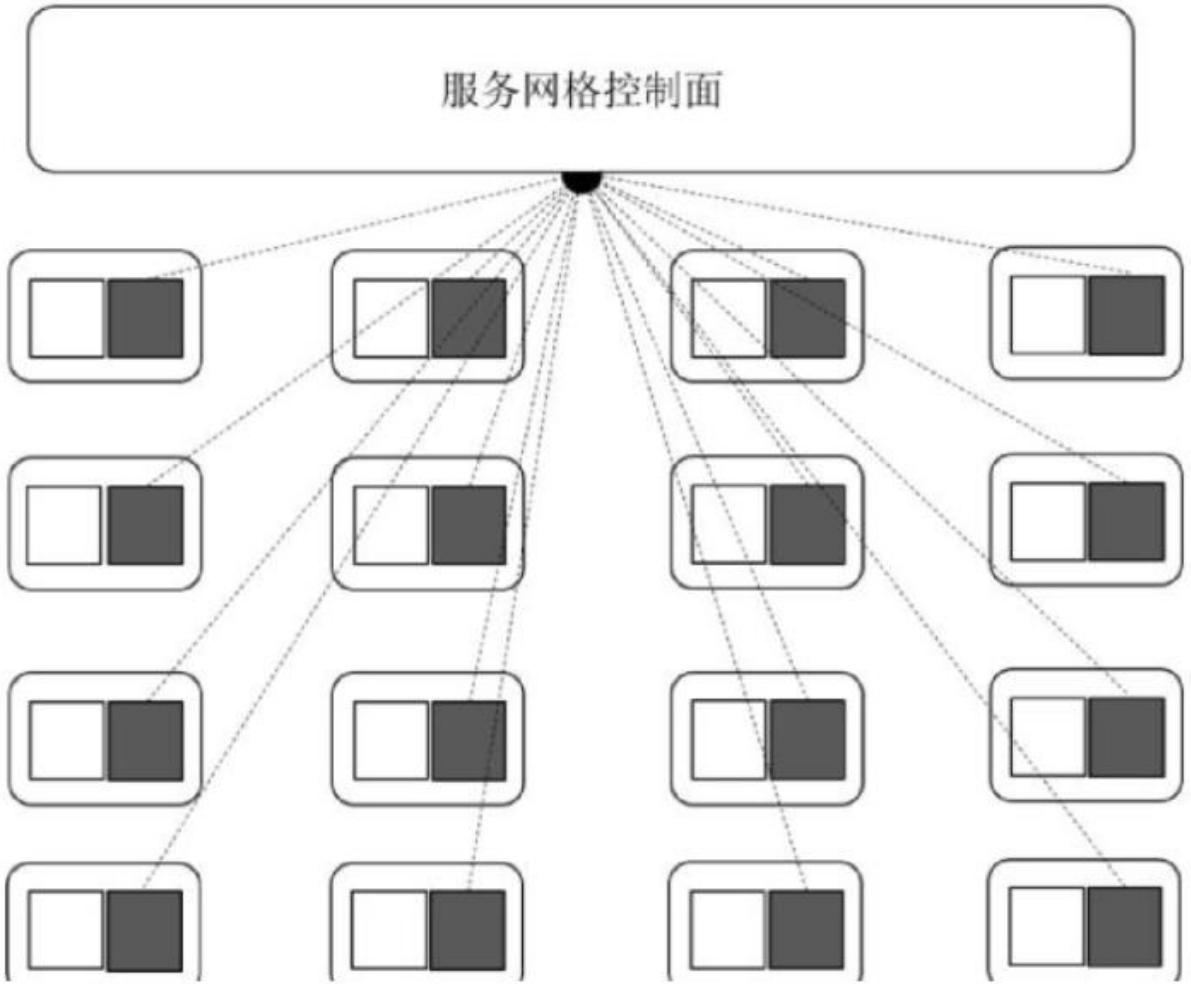
都需要工具的支撑

微服务



- 负载均衡
- 动态路由
- 灰度发布
- 故障注入
- 认证
- 鉴权
- 调用链
- 访问日志
- 监控

服务网格



- 无需考虑每种语言都要解决的问题
- 对业务代码0侵入，开发者无需关心分布式架构带来的复杂性以及引入的技术问题
- 对于不适合改造的老旧单体应用，提供了一种接入分布式环境的方式
- 微服务化的进程通常不是一蹴而就的，很多应用选择了演进的方式，就是将单体应用一部分一部分地进行拆分。而在这个过程中，使用Service Mesh就可以很好地保证未拆分的应用与已经拆分出来的微服务之间的互通和统一治理
- 开发出的应用既是云原生的又具有云独立性，不将业务代码与任何框架，平台或者服务绑定

云原生能带来什么

最新的调研显示:

- 87% 表示，云原生推动了更好的客户体验
- 84% 表示，云原生提高了收益并降低运维成本
- 80% 表示，新产品和服务的推行等待时间显着降低

云原生带来的改变

运维职能演变过程：

- 钱少事多受人欺

  运维部门是成本部门。有个词叫，成本优化。CXO 看到机器的负载这么低，就会想着裁撤机器，能少花就少花点，运维也就来活儿了。优化成本是运维的职责之一。

云原生带来的改变

运维职能演变过程：

- 虚拟容器拔地起

  基于虚拟机构建的 IT 基础设置，让运维人员摆脱了物理机房的束缚，不用花太多时间关注硬件。范围缩小了，运维质量也就上去了。

  Docker 的普及，意味着运维的春天不远了。虚拟化技术让运维不用关注硬件，Docker 让运维不用关注应用。运维只需要基于 Docker 展开运维工作即可，采集监控指标、日志，配置告警等。

云原生带来的改变

运维职能演变过程：

- 敏捷开发少不了

  敏捷开发发展于极限编程，是很重要的方法论。DevOps 被泛化，一千个人可能有一千个不同的解读，可以将其理解为一场 IT 文化运动。

  DevOps 正式出现，是在 2009 年。之后逐渐发展，历经十多年依然很受关注。

云原生带来的改变

运维职能演变过程：

- 站在枝头唧喳喳

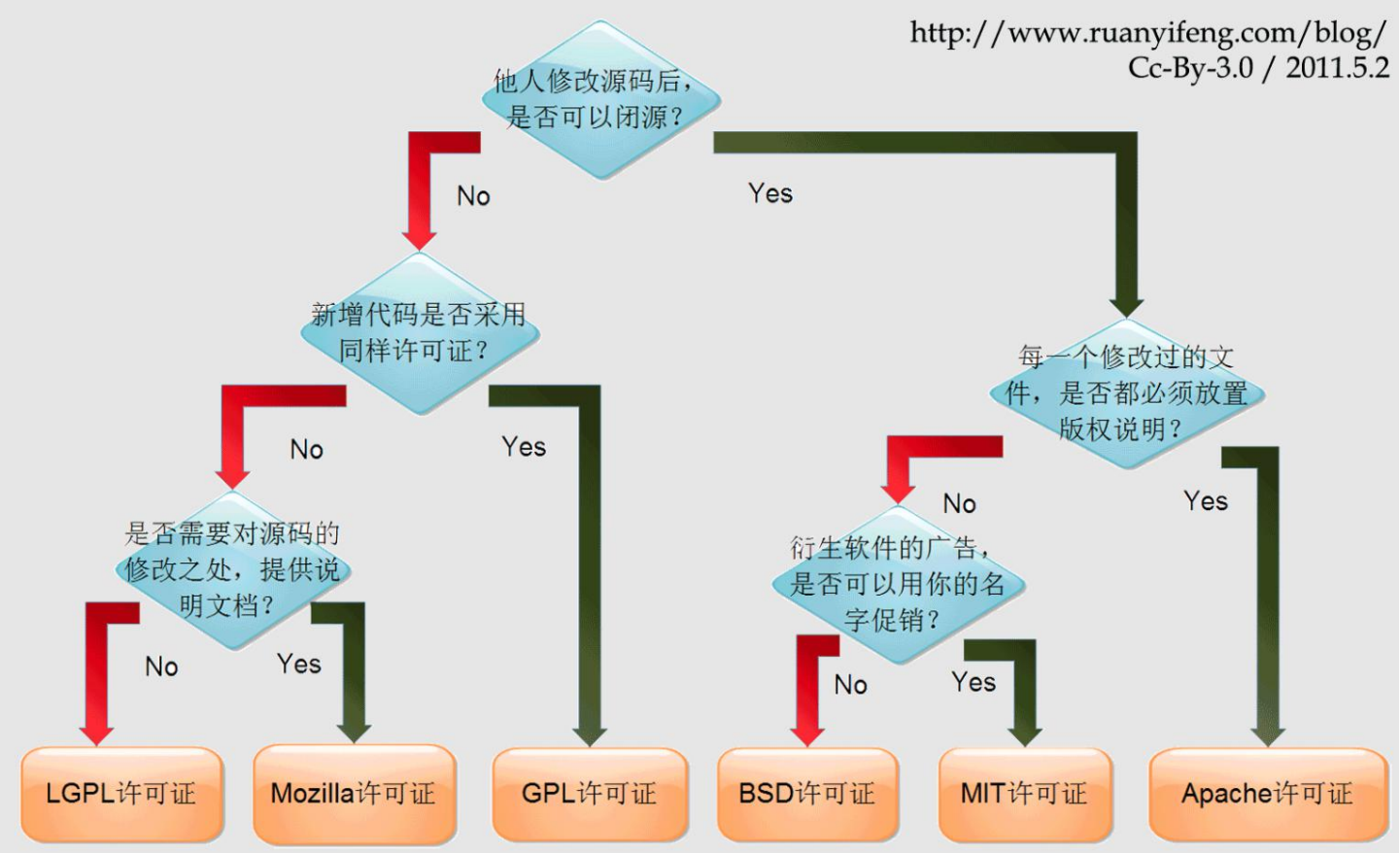
  从没见过哪个行业像 IT 行业这么热爱分享，更何况是一群以前没什么地位、大家都不愿意干的运维主导的。CNCF 每年都会举办云原生技术峰会 KubeCon。

  很多主题内容都是围绕 Kubernetes 进行的。Kubernetes 作为基础设置，替代了 OS\VM ，也就意味着之前所有的基础应用都需要往 Kubernetes 上迁移。除此，在使用 Kubernetes 的过程中，也会催生新的场景。

开源正在重构商业模式

- 1969 年，贝尔实验室将 Unix 代码共享给社区，为开源奠定了重要基础。
- 1984 年，Richard Stallman 离开 MIT，发起 GUN 项目，希望使用自由代码构建一个类 Unix 的操作系统。此后，Stallman 创立自由软件基金会（FSF），发起 GNU 通用公共协议证书（GNU General Public License, GNU GPL），开发了 GCC、Emacs 等一系列重要产品。
- 1991 年，Linus Torvalds 在 GPL 协议下发布了 Linux 操作系统内核。
- 1995 年，Apache 诞生，随即占据 Web 服务器大部分市场份额。
- 2000 年，开源延伸至移动和云领域，由 Google 等大企业驱动，影响技术发展路线和市场格局。
- 2008 年，Github 等代码托管平台，采用 pull/request 等方式协同合作，将开源推向了新高度。
- 2014 年，Google 开源 Kubernetes，获得极大关注。经过几年发展，Kubernetes 成为事实上的分布式架构平台。

开源协议



- 通常是 GPL V3
- 宽松的是 MIT、Apache
- 商业用 LGPL

开源的收益

- 更好的宣传

开源是非常有利的优势，代表着开放、先进的研发理念和技术。对于招聘人员来说，开源可以吸引更优秀的工程师

- 更好的开发实践

开源项目的代码质量更高，文档更完善，中断风险更小

- 更有利于整合上下游

通过开源，开发者还可以快速获取上下游的反馈信息，对产品进行调整、改进，有利于产品迭代。

开源的盈利方式 - 开源需要其他途径补贴

- 销售云服务补贴开源

SaaS 模式直接提供远程服务，计时计量收费，例如 Sentry

- 发行多版本

只提供了核心模块的源码，无法满足全部的企业需要，定制化

- 运维服务

各种运维场景，故障处理，层出不穷的安全漏洞和补丁解决方法，都可以成为付费点

- 培训、认证、授权
- 销售周边

销售公仔、文化衫、纪念品

国内比较活跃的开源组织

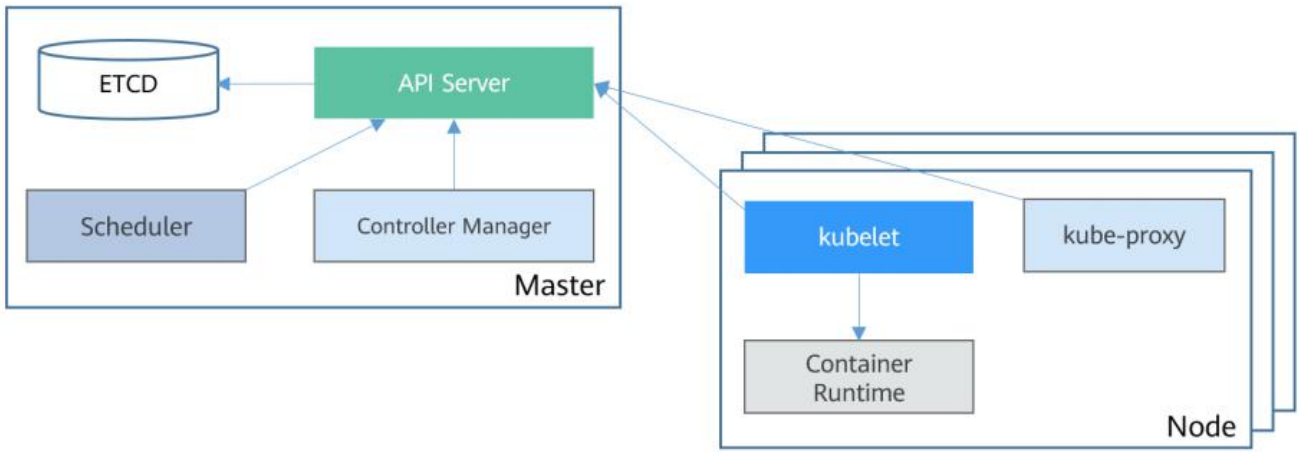
- PingCAP
  - TiDB
  - TiKV
- Huawei
  - KubeEdge
  - Karmada
- Aliyun
  - Kubevela
  - Dragonfly

Kubernetes

chenshaowen

目录

Kubernetes 架构



Master节点:

API Server 各组件互相通讯的中转站

Controller Manager 各种控制器

Scheduler 应用的调度

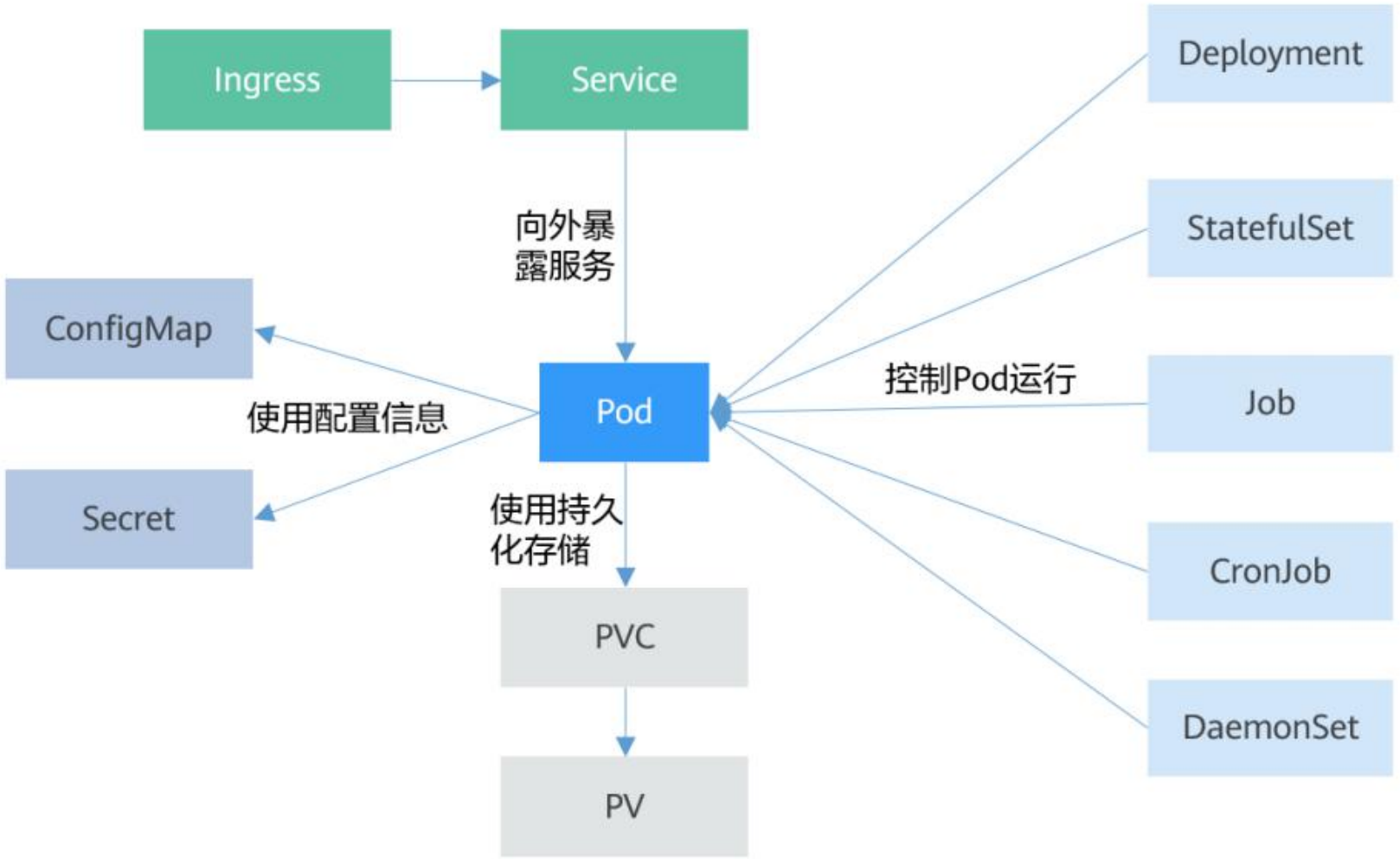
Node节点:

Kubelet 节点上容器生命周期

Kube-proxy 应用访问路由

Container Runtime 提供运行时

Kubernetes 基本对象



Kubernetes 内置了基本的对象，也允许开发者通过 CRD 的方式扩展对象。

Kubernetes 基本对象

- Pod 是 Kubernetes创建或部署的最小单位。

  一个Pod封装一个或多个容器 （container）、存储资源（volume）、一个独立的网络IP以及管理控制容器运行 方式的策略选项。

- Deployment 是对 Pod 的服务化封装。

  一个Deployment可以包含一个或多个Pod， 每个Pod的角色相同，所以系统会自动为Deployment的多个Pod分发请求。

- StatefulSet 是用来管理有状态应用的对象。

  和 Deployment 相同的是，StatefulSet 管理了基于相同容器定义的一组 Pod。但和 Deployment 不同的是，StatefulSet为 它们的每个Pod维护了一个固定的ID。这些Pod是基于相同的声明来创建的，但是 不能相互替换，无论怎么调度，每个Pod都有一个永久不变的ID。

- Job 是用来控制批处理型任务的对象。

  批处理业务与长期伺服业务 （Deployment）的主要区别是批处理业务的运行有头有尾，而长期伺服业务在用 户不停止的情况下永远运行。Job管理的Pod根据用户的设置把任务成功完成就自 动退出（Pod自动删除）。

Kubernetes 基本对象

- CronJob 是基于时间控制的 Job

  类似于Linux系统的crontab，在指定的时间周期 运行指定的任务。

- DaemonSet 是这样一种对象（守护进程）

它在集群的每个节点上运行一个 Pod，且保证只有一个Pod，这非常适合一些系统层面的应用，例如日志收集、资 源监控等，这类应用需要每个节点都运行，且不需要太多实例，一个比较好的例子就是Kubernetes 的 kube-proxy。

- Service Service是用来解决Pod访问问题的。

Service有一个固定IP地址，Service将访问流 量转发给Pod，而且Service可以给这些Pod做负载均衡。

- Ingress Service是基于四层TCP和UDP协议转发的，

Service是基于四层TCP和UDP协议转发的，Ingress可以基于七层的HTTP和HTTPS 协议转发，可以通过域名和路径做到更细粒度的划分。

Kubernetes 基本对象

- ConfigMap 是一种用于存储应用所需配置信息的资源类型

用于保存配置数据的 键值对。通过ÑnĒgD Ý可以方便的做到配置解耦，使得不同环境有不同的配 置。

- Secret是一种加密存储的资源对象

您可以将认证信息、证书、私钥等保存在 Secret中，而不需要把这些敏感数据暴露到镜像或者Pod定义中，从而更加安全和 灵活。

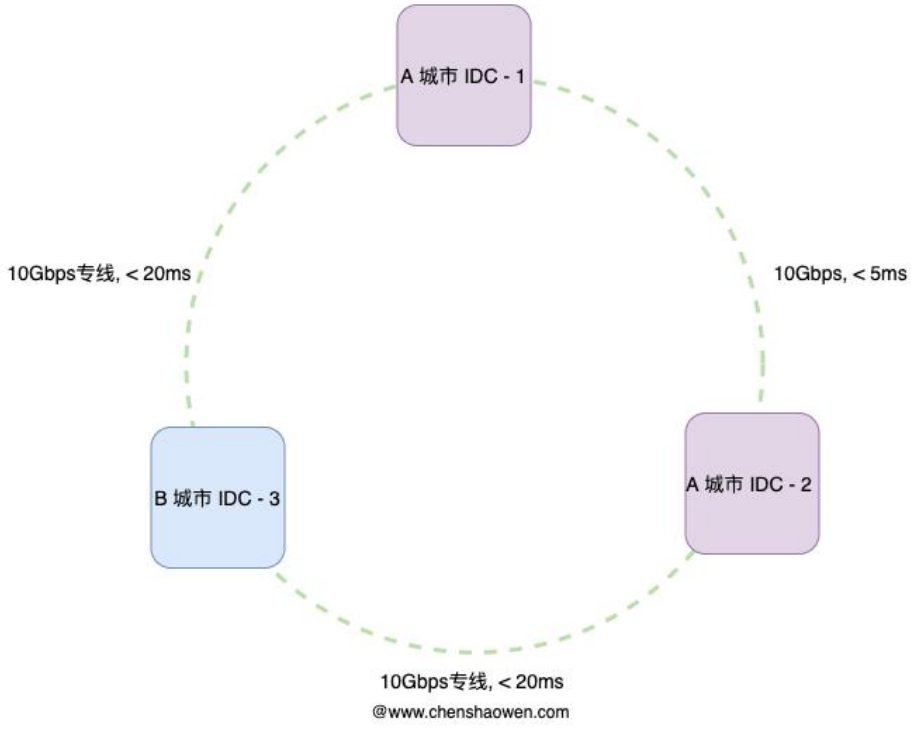
- PersistentVolume（PV） PV指持久化数据存储卷

主要定义的是一个持久化存储在宿主机上的目录，比如 一个NFS的挂载目录。

- PersistentVolumeClaim（PVC）

Kubernetes提供PVC专门用于持久化存储的申请，PVC可以让您无需关心底层存储 资源如何创建、释放等动作，而只需要申明您需要何种类型的存储资源、多大的 存储空间。

两地三中心的网络拓扑

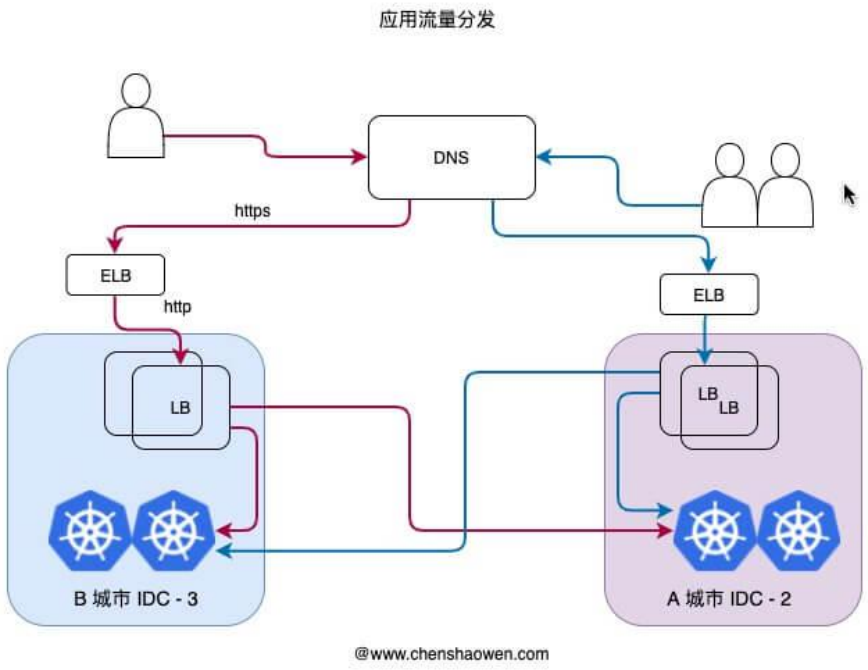


两地三中心架构的前提是，各个机房是互联互通的。

我们需要在机房间距、延时二者之间进行取舍:

- 机房间距离越远，容灾能力越强，但光纤会更长，延时会更高
- 机房间距离越短，容灾能力越差，但光纤会越短，延时会很低

应用流量的分发



1. 用户通过域名访问应用服务，通过智能 DNS 解析到地理上更近的机房 IP
2. 公有云的 ELB 会卸载 TLS 证书，并提供一定的安全防护功能。
3. 在机房中，使用虚拟机部署有一个 LB 服务，对流量进行切分，一部分流量被切分到了另外一个机房。两个机房的 LB 使用的是同一个存储后端。
4. 两个机房的服务，分别响应不同用户的请求

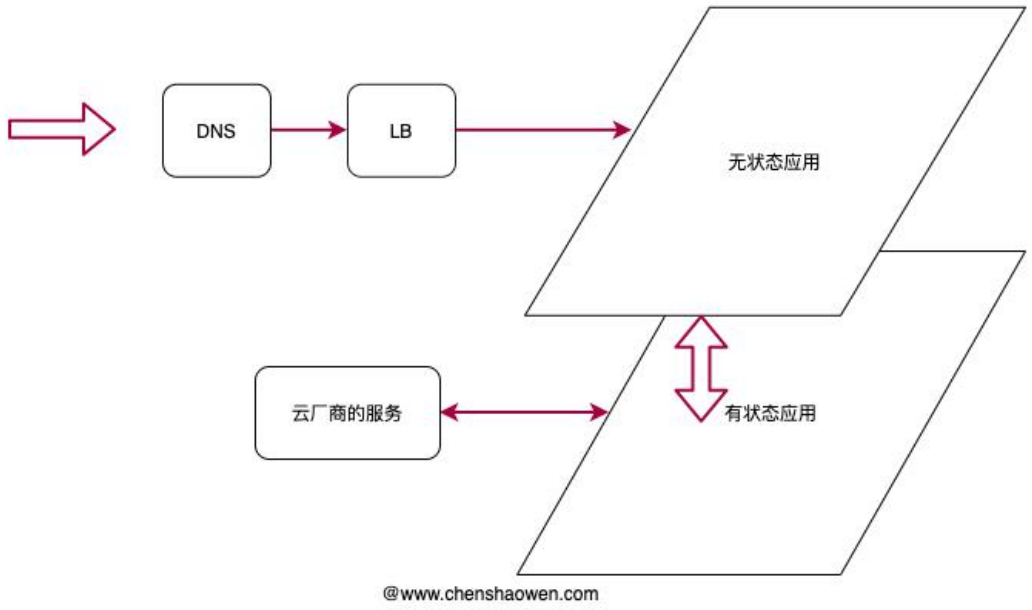
为什么是异地机房承载流量

没有经过流量冲击的路径是不可靠的。即使做了高可用、容灾，如果没有常态化的演练，系统也不会具备应对的能力。

因此，在多机房建设时，非常重要的一点就是，让更多机房获得访问流量。这里选择的是，两个异地的机房作为日常主要的流量机房，原因如下:

- 更好演练灾难发生时的状况
- 租用专线之后，异地机房延时能满足要求
- 有足够预算购买专线带宽

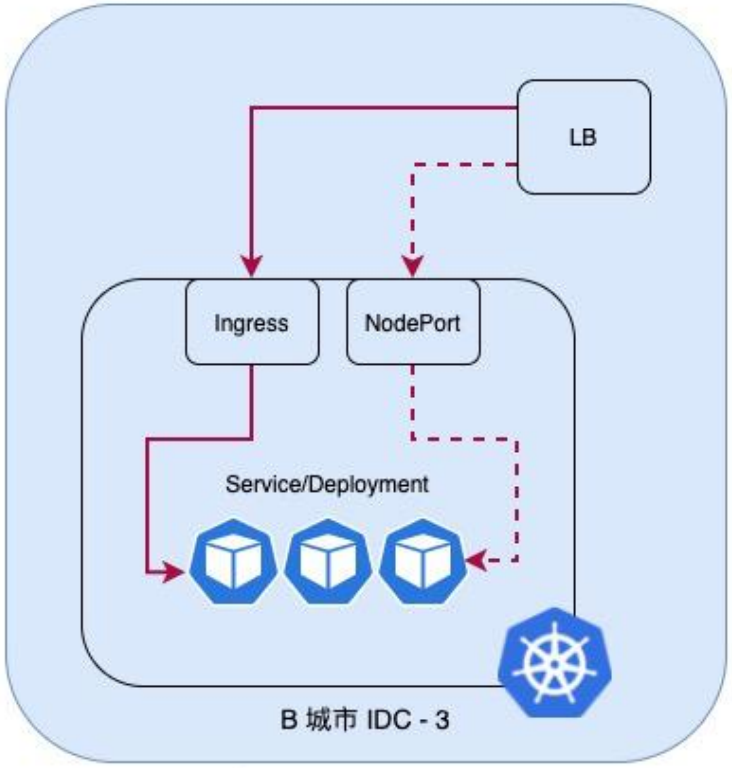
有状态与无状态的分层



有状态应用和无状态应用的分层，使得服务架构更加清晰。由无状态应用对外提供服务，而有状态应用为无状态应用提供服务。

这里的有状态应用，使用的是虚拟机部署的高可用服务，或者直接购买厂商的云服务中间件。

无状态应用



无状态应用基于 Kubernetes 提供运行时环境。得益于其强大的弹性与自愈能力，我们只需要关注于对各种云原生组件的使用，对参数的调优，即可满足大部分的业务需求。对于无状态应用，我们通常会采用 Ingress 或 NodePort 的方式，对外暴露服务。两者的区别主要在于:

- 支持的服务数量。每个 NodePort 会占用一个端口
- 功能差异。Ingress 能提供 Host、灰度、子 Path 路径等功能
- 组件数量。Ingress 需要更多组件支撑
- 运维成本。Ingress 更新时，影响面更大，运维成本高
- 迁移成本。NodePort 可能会发生端口冲突

Kubernetes 并不是保证服务 100% 可用，而是一旦服务异常时，能够快速利用空闲资源新建。同时，Kubernetes 还面临集群升级、主机维护等问题，因此，对于一些低频变更、对稳定性要求高的服务，我们采用的是虚拟机部署。

比如这里的 LB，LB 是一个影响面很大的应用，而且数量不会很多，我们通常会采用高可用的模式，部署在几台虚拟机上。

Kind 集群

- 什么是 Kind

Kind 是另一个Kubernetes SIG项目，但它与minikube 有很大区别。它可以将集群迁移到 Docker 容器中，这与生成虚拟机相比，启动速度大大加快。简而言之，kind是一个使用Docker容器节点运行本地Kubernetes集群的工具（CLI）。

- 安装 kubectl 命令

https://kubernetes.io/zh/docs/tasks/tools/

- 安装 Kind 命令

go get sigs.k8s.io/kind@v0.8.1

随堂实验 01 - 使用 Kind 部署一个集群

```
Usage:
  kind [command]

Available Commands:
  build       Build one of [node-image]
  completion  Output shell completion code for the specified shell (bash, zsh or fish)
  create      Creates one of [cluster]
  delete      Deletes one of [cluster]
  export      Exports one of [kubeconfig, logs]
  get         Gets one of [clusters, nodes, kubeconfig]
  help        Help about any command
  load        Loads images into nodes
  version     Prints the kind CLI version

Flags:
  -h, --help              help for kind
      --loglevel string   DEPRECATED: see -v instead
  -q, --quiet             silence all stderr output
  -v, --verbosity int32   info log verbosity
      --version           version for kind
```

kind create cluster

Kubernetes 基本命令

- 查看节点

kubectl get node -o wide --show-labels

- 创建一个 Nginx 服务

kubectl run deploy nginx --image=nginx

- 暴露一个服务

kubectl expose deploy nginx --type=NodePort --port=80 --target-port=80

- 查看部署和服务

kubectl get pod,deploy,svc -o wide

问答

- Kubernetes 包含哪些集成的组件
- 使用 kubectl create 命令创建一个 Deployment 会有哪些过程

随堂实验 02 - 在 Kind 集群上部署一个服务

1. 拉取一个 Nginx 镜像
2. 将服务运行在 Kubernetes 上
3. 导出服务的 Kubernetes Yaml