模块三：容器和 K8s 入门

王炜/ 前腾讯云 CODING 高级架构师

目录

1 深入 Dockerfile 和镜像构建

2 初级应用定义：Manifest

3 部署示例应用

4 高级应用定义：Helm、Kustomize

5 Helm、Kustomize 实战

1. 深入 Dockerfile 和镜像构建

容器历史

极客时间 训练营

2013

Docker

最初使用 LXC Linux Containers 实现，后来用 libcontainer 取代

2008

LXC Linux Containers

使用 namespace 和 cgroups 实现的第一个容器管理

2007

Linux Control Group

Process Containers 重命名并合并到 Kernel 2.6.24

2006

Google Process Containers

资源隔离（CPU、内存、IO、网络）

2004

Solaris Zones

快照、克隆

2002

Linux Namespaces

进程隔离

2000

FreeBSD Jails

早期的容器技术

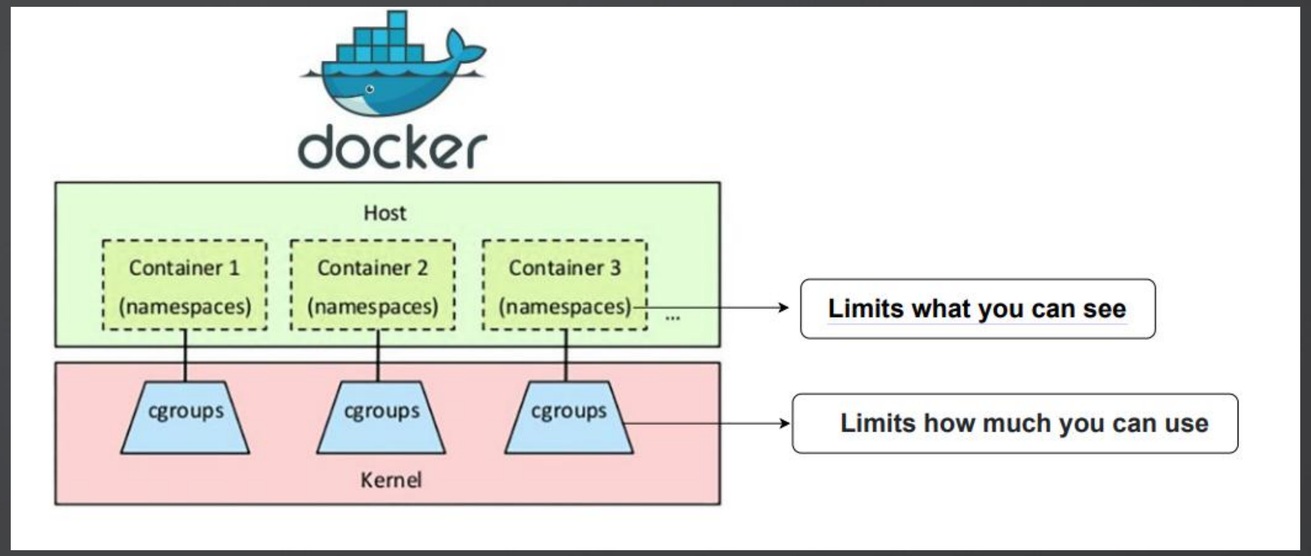
1979

Chroot

文件系统隔离

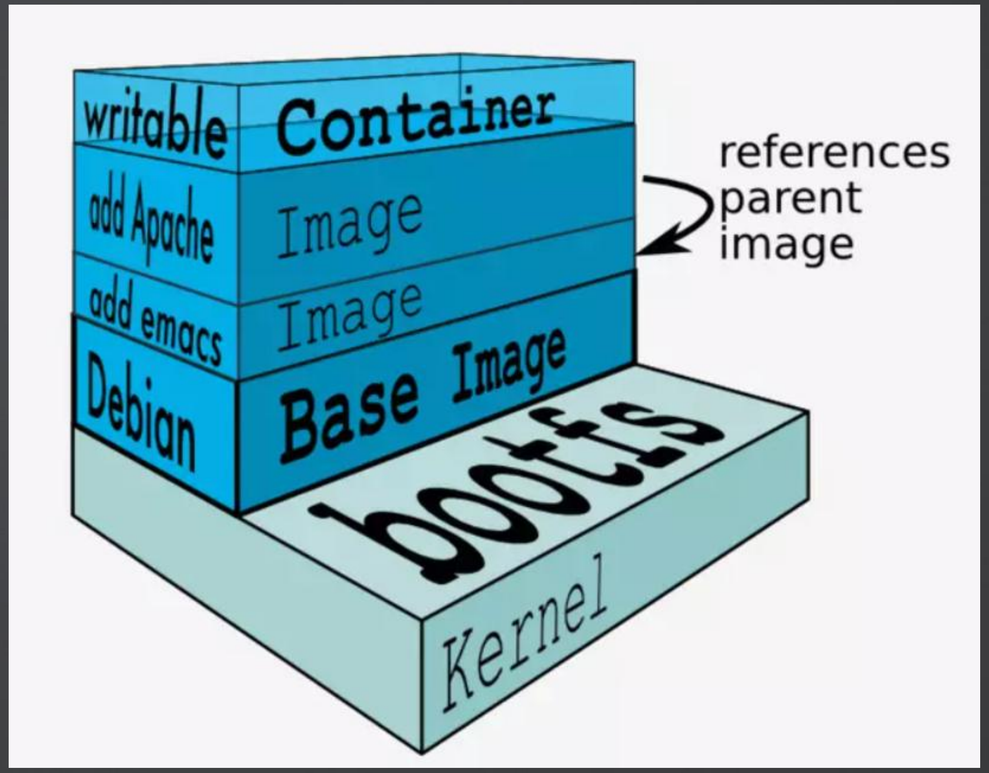
容器技术的核心原理

- cgroups：限制进程的 CPU、内存、网络、IO 等
- namespace：隔离进程，为进程提供独立的文件系统、网络、主机名等，有以下类型：
  - Mount：隔离文件系统
  - PID：隔离进程
  - Network：隔离网络
  - User：隔离用户和组
  - IPC：进程间通信隔离
  - UTS：UNIX 分时，例如主机名和域



怎么理解容器和镜像?

- 镜像是容器的定义，包括文件系统、环境变量以及默认的启动命令
- 容器是是镜像的实例，使用同一个镜像启动的容器是相互隔离的，不同的容器之间也是相互隔离的
- 有点类似于编程里的 interface 和 class，一个负责定义，一个负责实现



不可变特性

极客时间 训练营

- 容器是不可变的（在不同环境有相同表现）
- 容器被设计为一次性（临时的）
- 新启动的容器是镜像最初始状态
- 数据被存储在容器外部

什么是 Dockerfile?

- 一组指令的描述文件，执行这些命令可以在某一个基础镜像的基础上创建新的 Docker 镜像
- Docker 可以通过读取 Dockerfile 构建镜像
- 用户通过 docker build 命令启动镜像构建过程

构建镜像

极客时间 训练营

- 编写 Dockerfile

```
FROM debian:latest
RUN apt-get update && apt-get install nginx -y
CMD ["nginx", "-g", "daemon off;"]
```

- 构建镜像

```
docker build -t nginx -f Dockerfile .
```

构建上下文："." 代表当前目录，Docker 会将上下文的内容传输到 Docker daemon，使用 ADD 或 COPY 会将上下文的内容复制到镜像中。还可以使用 .dockerignore 来忽略特定目录，如项目中的node_module 目录，提高构建速度。

动手实践：隔离机制

极客时间 训练营

- 运行 nginx 镜像，体验隔离机制
- sudo docker run –d nginx:latest
- sudo docker ps
- sudo docker inspect ––format {{.State.Pid}} container_id　#获取容器在宿主机的进程 Pid
- ps –aux | grep 27725　# 查看宿主机进程详情
- sudo lsns ––task 27725　# 获取 PID 所有 namespace 类型
- sudo nsenter ––target 27725 ––mount # 进入 filesystem namespace，无需 docker daemon
- ls && whoami
- hostname my–containers　&& exit && hostname # 修改容器的主机名，同时会修改宿主机的主机名，这是因为没有进入到 utc namespace
- sudo nsenter ––target 27725 ––all # 进入所有 namespace
- hostname my–containers2　&& exit && hostname # 再次修改 hostname，退出后对比宿主机 hostname

深入镜像本质

极客时间 训练营

- 联合文件系统是镜像的基础
- 镜像是多层 Layer 的堆叠
- 构建时通过叠加新的 Layer 来实现更改
- Layer 在构时会被缓存
- Layer 以 hash 命名

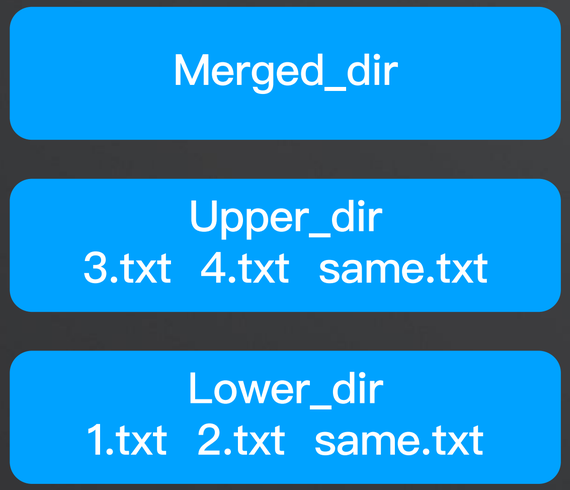
```
# Export filesystem of a container image as a tarball
$ docker build –t nginx:v1 –f iac/week–2/demo–1/Dockerfile .
$ docker save nginx:v1 –o ~/Downloads/nginx.tar
```

两层 Layer，Layer 名称以 Layer 内容的 hash 命名，对 Layer.tar 解压之后可见原始文件系统。

第一层 Layer 是基础镜像，第二层是安装 nginx 带来的变更（/usr/sbin、/var/www、/var/log）

层是怎么叠加，如何工作?

动手实践：联合文件系统（Overlay）




```
# 在 VM 中执行
$ cd /tmp/overlay
$ ls lower_dir && ls upper_dir && ls merged_dir
# lower_dir 是只读层，upper_dir 是可读写的，merged_dir 是用户最终看到的目录，work_dir 存储中间结果
,# 创建 OverlayFS
$ sudo mount -t overlay -o lowerdir=lower_dir/,upperdir=upper_dir/,workdir=work_dir/ none merged_dir/
$ df -a # 查看挂载的文件系统
$ ls merged_dir/
1.txt  2.txt  3.txt  4.txt  same.txt
$ cat merged_dir/same.txt  # 上层 Layer 会覆盖下层 Layer 的同名文件
$ vi merged_dir/1.txt  # 修改 overlay 合并层中位于 lower_dir 的 1.txt 文件
$ cat lower_dir/1.txt # 仍然是空内容，不会对 lower_dir 产生修改
$ cat upper_dir/1.txt  # 但在 upper_dir 里新增加了之前没有的 1.txt 文件，这就是“写时复制”的功能
$ rm merged_dir/2.txt
$ ls merged_dir/  # 文件被删除了
$ ls lower_dir/    # 在底层的 Layer 文件并没有被删除
$ ls -al upper_dir/  # 出现了 2.txt，但被标记为了 C，代表该文件已删除
$ ls -al merged_dir/  # 在合并层不显示 2.txt 的删除标记，而是显示完整合并之后的内容
# Lower_dir 类比镜像，Upper_dir 就像是容器启动之后覆盖的可写层
```

动手实践：Overlay 和 Docker 镜像

极客时间 训练营

```
$ docker image inspect nginx:latest | jq –r '.[0] | {Data: .GraphDriver.Data}'

{

  "Data": {

    "LowerDir": "/var/lib/docker/overlay2/195070a290d2fc358ca0a9e34c7f98b0c0e1e7eeb39b4d67501f03d3f686b304/diff"，# 可以挂载多个 LowerDir（:分割）

    "MergedDir": "/var/lib/docker/overlay2/q5gr38rehs74av5an85d9qchp/merged",

    "UpperDir": "/var/lib/docker/overlay2/q5gr38rehs74av5an85d9qchp/diff",

    "WorkDir": "/var/lib/docker/overlay2/q5gr38rehs74av5an85d9qchp/work"

  }

}
```

Dockerfile 指令

极客时间 训练营

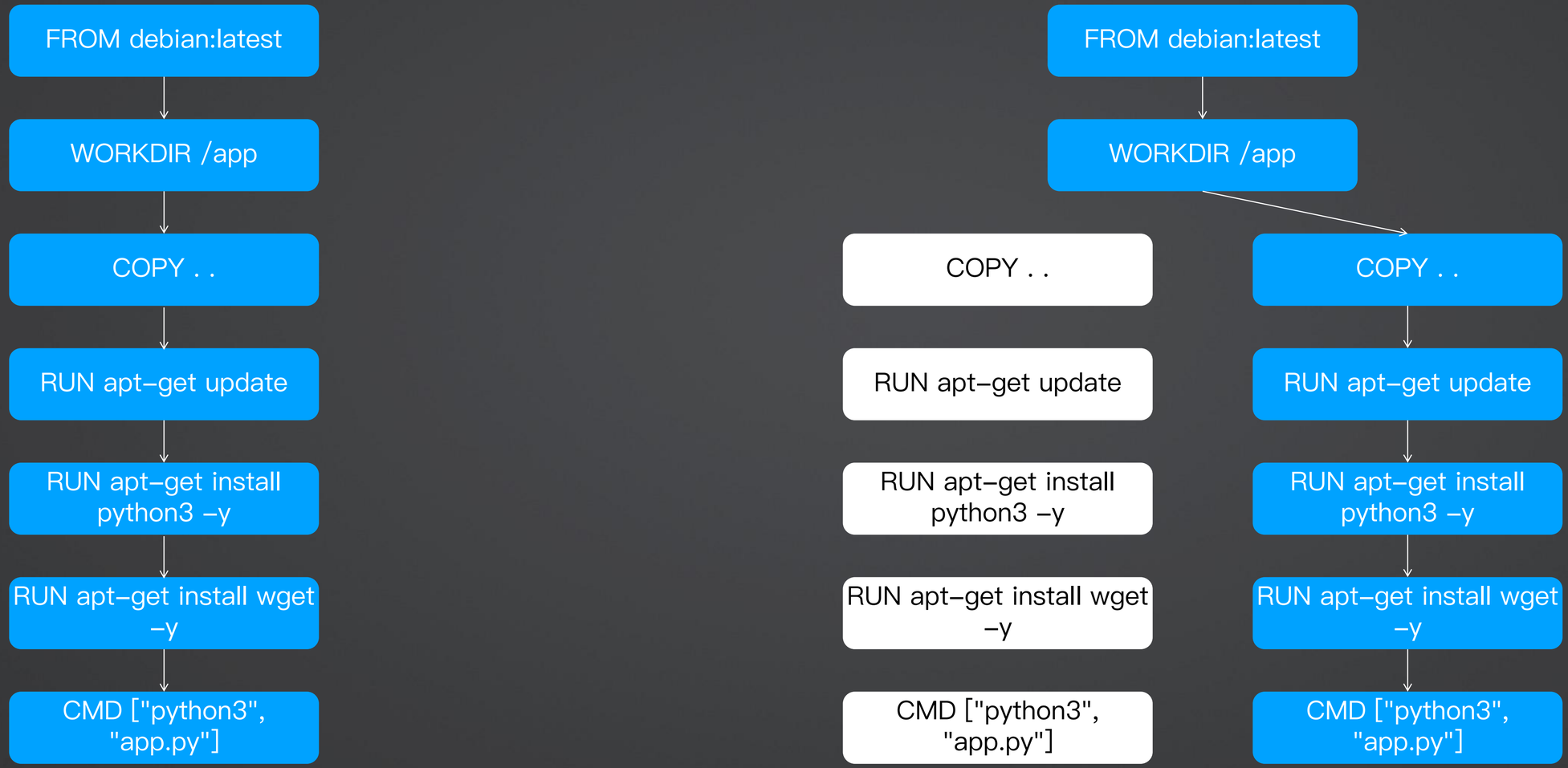
- FROM
- RUN
- COPY
- ADD
- EXPOSE
- CMD
- ENTRYPOINT

从 Layer 特性看 Dockerfile 的优化

极客时间 训练营

```
FROM debian:latest


WORKDIR /app

COPY . .


RUN apt-get update

RUN apt-get install python3 -y

RUN apt-get install wget -y


CMD ["python3", "app.py"]
```

为什么会无法使用缓存?


FROM debian:latest
WORKDIR /app
COPY . .
RUN apt-get update
RUN apt-get install python3 -y
RUN apt-get install wget -y
CMD ["python3", "app.py"]
FROM debian:latest
WORKDIR /app
COPY . .
RUN apt-get update
RUN apt-get install python3 -y
RUN apt-get install wget -y
CMD ["python3", "app.py"]
COPY . .
RUN apt-get update
RUN apt-get install python3 -y
RUN apt-get install wget -y
CMD ["python3", "app.py"]

从 Layer 特性看 Dockerfile 的优化

极客时间 训练营

```
FROM debian:latest


WORKDIR /app

RUN apt-get update && apt-get install python3 -y && apt-get install wget -y    # 减少 Layer 层数

COPY . .            # 将经常会产生变化的层移靠后

CMD ["python3", "app.py"]
```

Dockerfile 最佳实践

极客时间 训练营

- 减少层数
- 注意语句顺序
- 使用 .dockerignore
- 减小镜像体积
  - 选择合适的基础镜像（Alpine、slim 等）
  - 使用多阶段构建
- 安全
- 其他

注意语句顺序：依赖定义文件

极客时间 训练营

```
FROM golang:1.19

WORKDIR /app


COPY go.mod go.sum ./      # 先复制依赖定义文件，并安装依赖

RUN go mod download


COPY *.go ./                    # 再复制源码
```

使用 .dockerignore

极客时间 训练营

```
**/node_modules/
**/dist
.git
npm-debug.log
.coverage
.coverage.*
.env
.aws
```

减小镜像体积：选择合适的基础镜像

极客时间 训练营

镜像	大小
node:latest	380M
node:slim	76M
node:current-alpine3.18	52M

慎用 Alpine 镜像

极客时间 训练营

Ubuntu/Debian	Alpine
GNU C library	musl libc
丰富的第三方包	第三方包较少

减小镜像体积：使用多阶段构建

```
# syntax=docker/dockerfile:1
FROM golang:1.17
WORKDIR /opt/app
COPY . .
RUN go build -o example
CMD ["/opt/app/example"]
```

903M

```
# syntax=docker/dockerfile:1
# Step 1: build golang binary
FROM golang:1.17 as builder
WORKDIR /opt/app
COPY . .
RUN go build -o example

# Step 2: copy binary from step1
FROM ubuntu:latest
WORKDIR /opt/app
COPY --from=builder /opt/app/example ./example
CMD ["/opt/app/example"]
```

75M

安全：避免使用 root 用户

极客时间 训练营

```
FROM debian

RUN useradd –ms /bin/bash app

USER app

WORKDIR /app
```

其他：固定基础镜像的 Tag

极客时间 训练营

```
FROM ubuntu:23.10


FROM ubuntu

FROM ubuntu:latest
```

其他：尽量使用官方基础镜像

```
FROM python:slim-bullseye

FROM someone/python:latest
```

python DOCKER OFFICIAL IMAGE · 1B+ · ☆8.9K

Python is an interpreted, interactive, object-oriented, open-source programming language.

其他：设置时区

```
# Apline
ENV TZ Asia/Shanghai

RUN apk add tzdata && cp /usr/share/zoneinfo/${TZ} /etc/localtime \
    && echo ${TZ} > /etc/timezone \
    && apk del tzdata
```

```
# Debian
ENV TZ=Asia/Shanghai \
    DEBIAN_FRONTEND=noninteractive

RUN ln -fs /usr/share/zoneinfo/${TZ} /etc/localtime \
    && echo ${TZ} > /etc/timezone \
    && dpkg-reconfigure --frontend noninteractive tzdata \
    && rm -rf /var/lib/apt/lists/*
```

延伸：镜像构建、DinD、buildkit

极客时间 训练营

```
$ docker build –t app –f Dockerfile .
```

延伸：镜像构建、DinD、buildkit

极客时间 训练营

```
$ docker build –t app –f Dockerfile .
```

```
$ docker run –v /var/run/docker.sock:/var/run/docker.sock –it docker
```

延伸：镜像构建、DinD、buildkit

```
$ docker build –t app –f Dockerfile .
```

```
$ docker run –v /var/run/docker.sock:/var/run/docker.sock –it docker
```

```
buildctl build \
    ––frontend=dockerfile.v0 \
    ––local context=. \
    ––local dockerfile=. \
    ––output type=image,name=app:latest
# https://github.com/moby/buildkit
```

https://github.com/GoogleContainerTools/kaniko

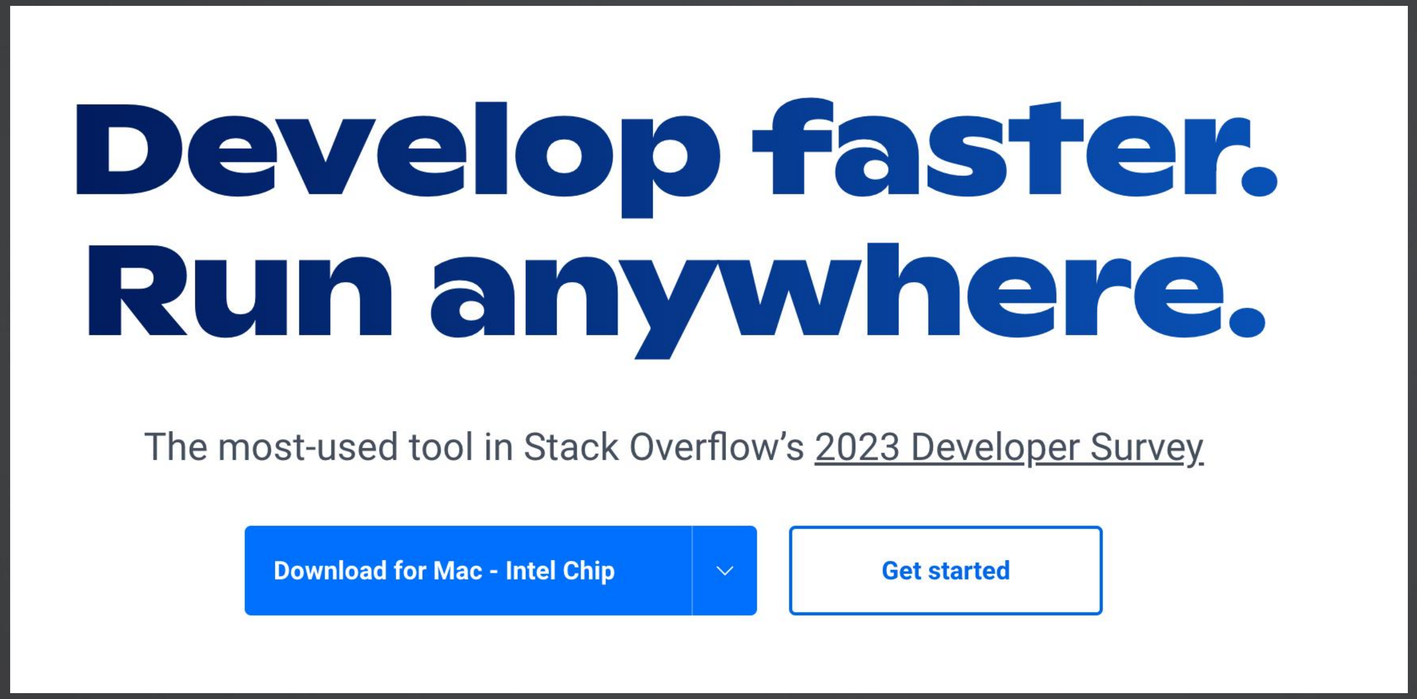
Demo

极客时间 训练营

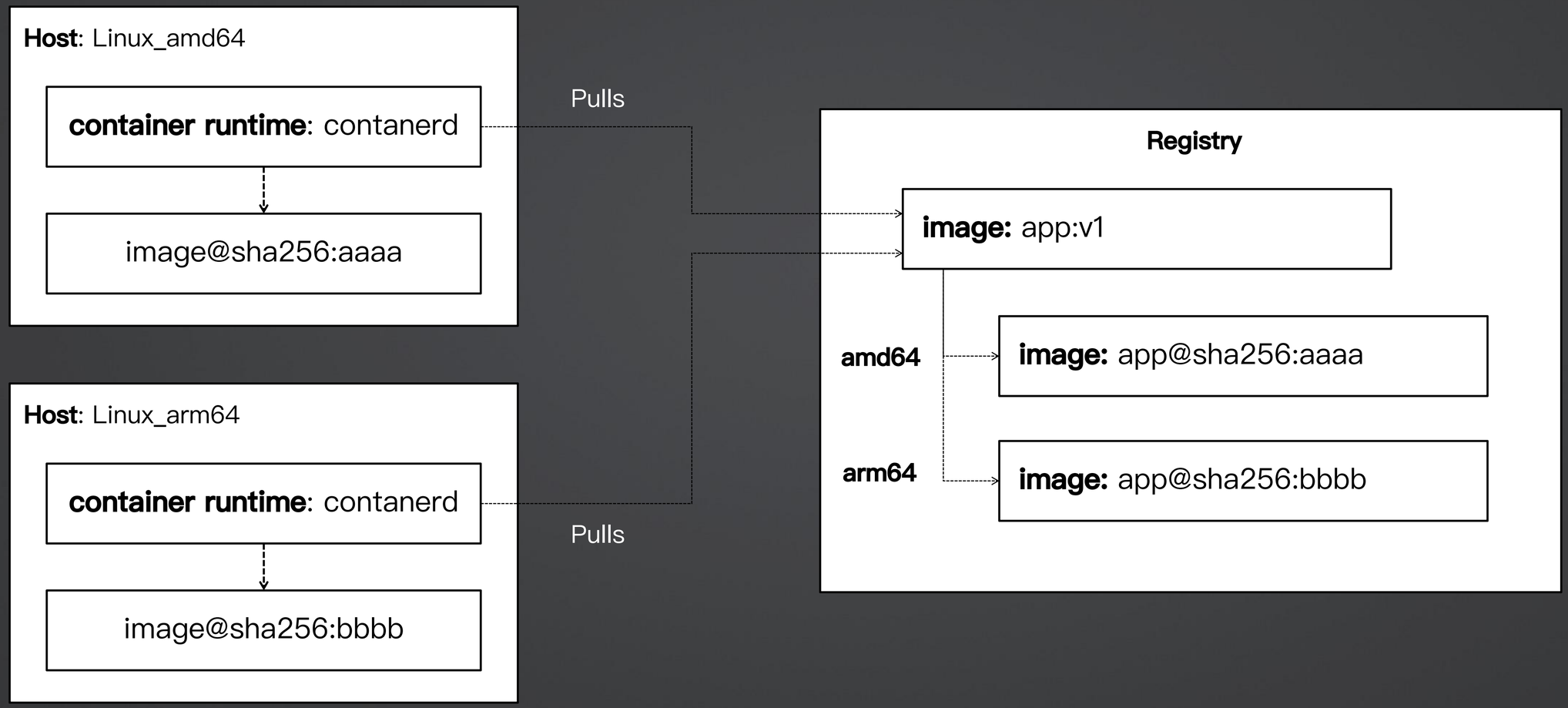
以下 golang 代码，要求构建出尽量小的镜像

```
package main

import "fmt"

func main() {

	fmt.Println("hello world")

}
```

思考：一次构建，到处运行？


Develop faster.
Run anywhere.
The most-used tool in Stack Overflow's 2023 Developer Survey
Download for Mac - Intel Chip
Get started

思考：一次构建，到处运行？


Host: Linux_amd64
container runtime: contanerd
image@sha256:aaaa
Pulls
Registry
image: app:v1
amd64
image: app@sha256:aaaa
arm64
image: app@sha256:bbbb
Host: Linux_arm64
container runtime: contanerd
image@sha256:bbbb
Pulls

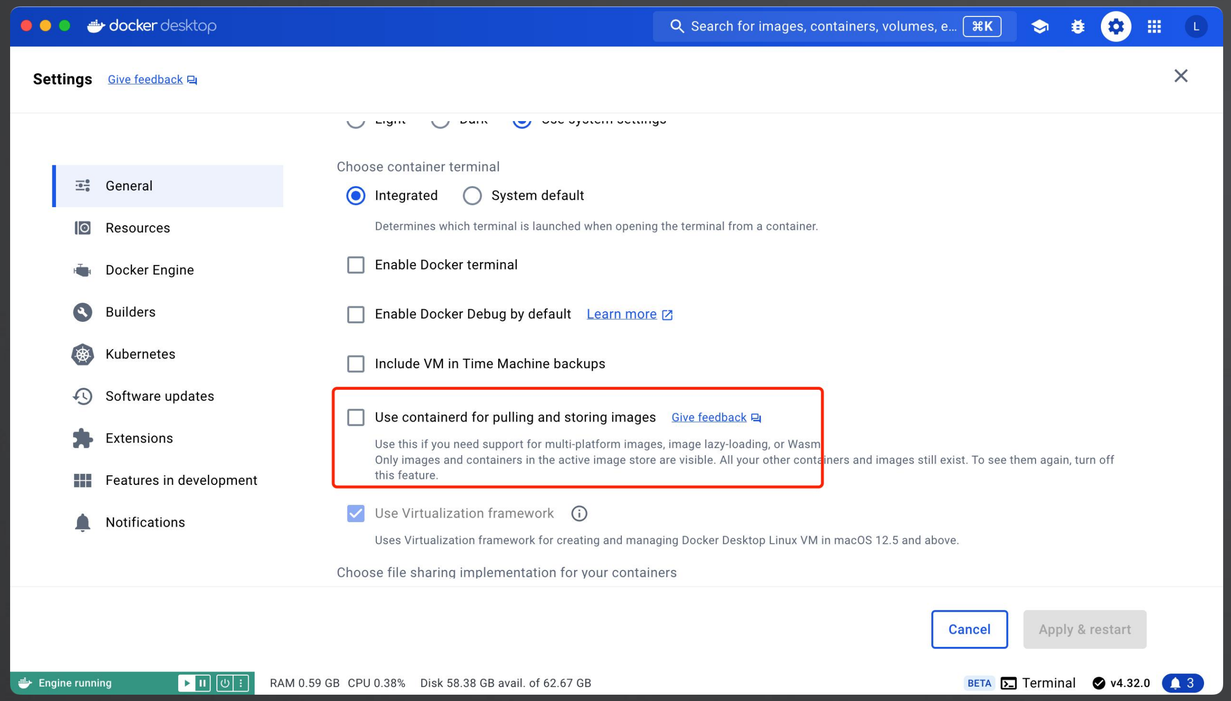
思考：一次构建，到处运行？

极客时间 训练营

```
$ crane manifest alpine:3.18.3 | jq .                    # 本地执行，查看镜像的 Manifest

$ crane manifest alpine:3.18.3@sha256:c5c5fda71656f28e49ac9c5416b3643eaa6a108a8093151d6d1afc9463be8e33 | jq .
                                                         # 查看 amd64 平台镜像的 Manifest

$ mkdir tmp && cd tmp && crane export –v alpine:3.18.3 – | tar xv        # 在临时目录下执行，查看镜像、manifest 拉取的具体过程

$ ls
```

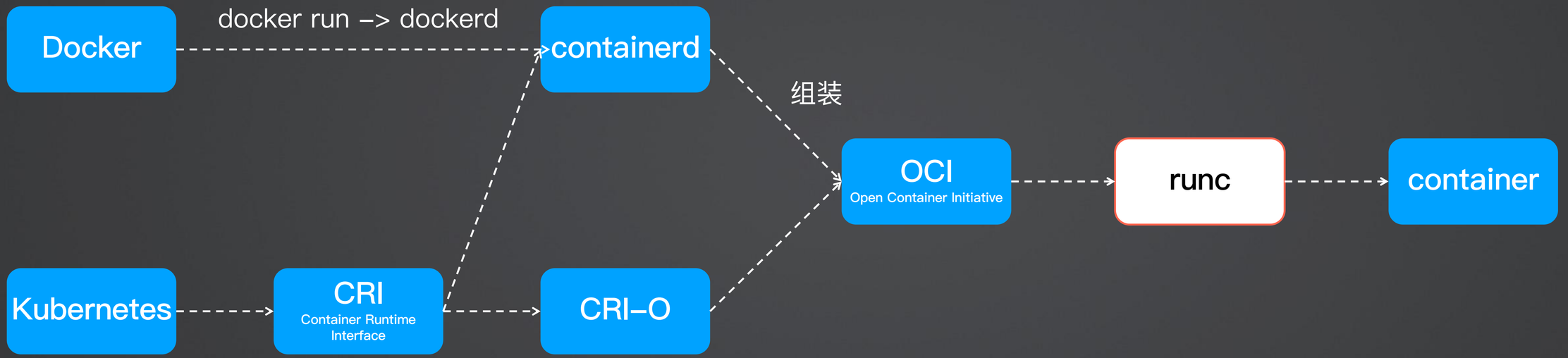
构建多平台镜像

```
# 一次性构建多平台镜像，并推送到镜像仓库
$ docker build --platform linux/amd64,linux/arm64 -t test:v1 -f Dockerfile-1 .
```

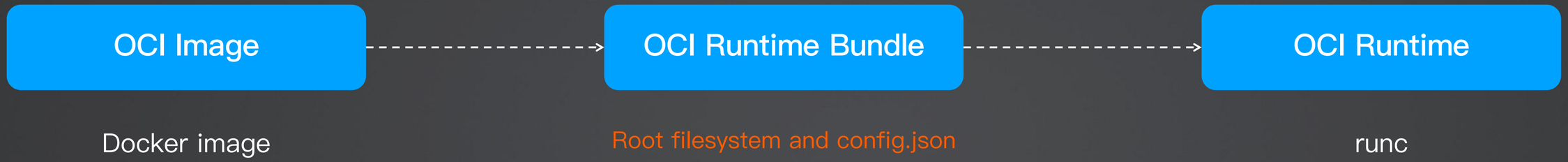


https://docs.docker.com/build/building/multi-platform/#qemu

Docker、Containerd、CRI-O、runc



runc



https://github.com/opencontainers/runtime-spec/blob/main/config.md

动手实践：使用 runc 直接启动容器

```
# VM 里运行
$ cd /tmp
$ ls rootfs                          # 提前准备好的 busybox rootfs
$ runc spec                          # 生成一个 config.json 样例
$ cat config.json
$ sudo runc run my-container         # 通过 runc 直接启动容器
$ ps aux
$ hostname
$ exit                               # 退出容器
```

2. 初级应用定义：Manifest

Manifest

极客时间 训练营

- 工作负载
  - Deployment
  - StatefulSet
  - DaemonSet
  - Jobs
  - CronJob
- Service
  - ClusterIP
  - NodePort
  - Loadbalancer
  - Headless
- Ingress
- 配置
  - ConfigMap
  - Secret
- HPA
- 存储
  - StorageClass
  - PV、PVC

https://kubernetes.io/zh-cn/docs/concepts/workloads/controllers/deployment/

指南1：K8s YAML 难记

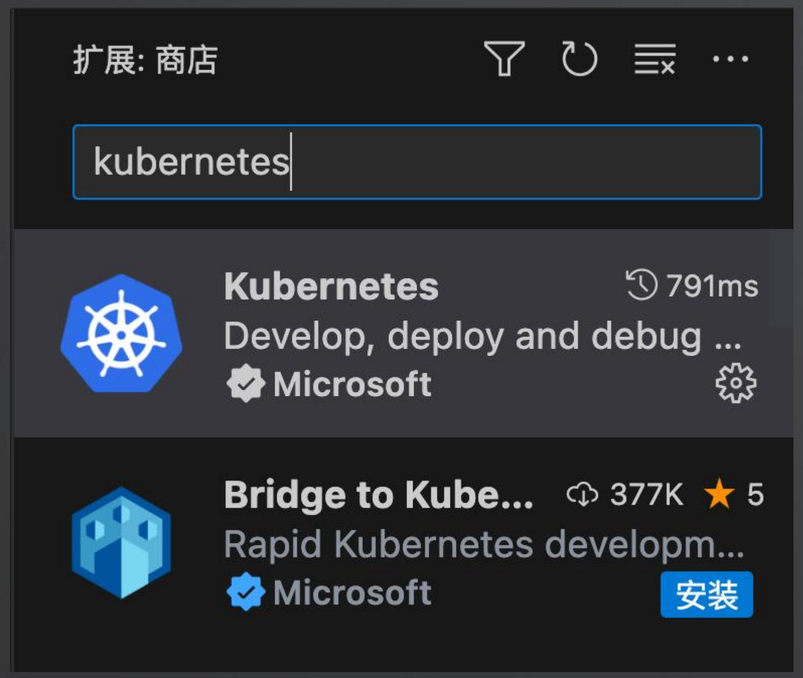
极客时间 训练营

```
# 使用 kubectl create 命令来生成 YAML 模板

$ kubectl create deployment nginx --image=nginx -o yaml --dry-run=client

$ kubectl create service clusterip my-cs --tcp=5678:8080 -o yaml --dry-run=client

$ kubectl create configmap my-config --from-literal=key1=config1 --from-literal=key2=config2 -o yaml --dry-run=client

# 使用 --help 来查看创建参数

$ kubectl create [workload|config|storage] --help
```

指南1：K8s YAML 难记

极客时间 训练营

```
# 安装 VSCode Kubernetes 插件
https://marketplace.visualstudio.com/items?itemName=ms-kubernetes-tools.vscode-kubernetes-tools
```



指南2：YAML 多文件合并

极客时间 训练营

```
apiVersion: v1
kind: Service
metadata:
  name: myapp
---
apiVersion: v1
kind: Service
metadata:
  name: myapp-2
```

指南3：注意 YAML 的“挪威问题”

极客时间 训练营

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp
spec:
  containers:
  - name: myapp
    image: busybox
    env:
      - name: GERMANY
        value: DE
      - name: NORWAY
        value: NO          # NO 会被转化为布尔值，需要使用引号包起来
```

https://yaml.org/type/bool.html

保留关键字：y|Y|yes|Yes|YES|n|N|no|No|NO
|true|True|TRUE|false|False|FALSE
|on|On|ON|off|Off|OFF

指南4：注意数字类型

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp
spec:
  containers:
  - name: myapp
    image: busybox
    env:
      - name: GO_VERSION
        value: 1.10                # 1.10 会被转化为数字 1.1，需要用引号包起来
```

https://yaml.org/type/float.html

指南4：多行内容

极客时间 训练营

```
key: >
  Your long
  string here.
key: >-
  Your long
  string here.
key: |
  ### Heading
  * Bullet
  * Points
key: |-
  ### Heading
  * Bullet
  * Points
```

思考：4 种写法，他们有什么区别?

指南4：多行内容

```
key: >                # 不保留文字间的换行，保留末尾的换行
   Your long
   string here.
key: >-               # 不保留文字间的换行，不保留末尾的换行
   Your long
   string here.
key: |                # 保留文字间的换行，保留末尾的换行
   ### Heading
   * Bullet
   * Points
key: |-               # 保留文字间的换行，不保留末尾的换行
   ### Heading
   * Bullet
   * Points
```

```
Your long string here.(\n)
---
Your long string here.
---
### Heading
  * Bullet
  * Points(\n)
---
### Heading
  * Bullet
  * Points
```

指南4：多行内容

- >, |: "clip": 保留换行，删除尾部空行
- >–, |–: "strip": 删除换行符，删除尾部空行
- >+, |+: "keep": 保留换行，保留尾部空行

指南5：使用引用

极客时间 训练营

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  selector:
    matchLabels: &pod-labels
      app: myapp
  template:
    metadata:
      labels: *pod-labels
```

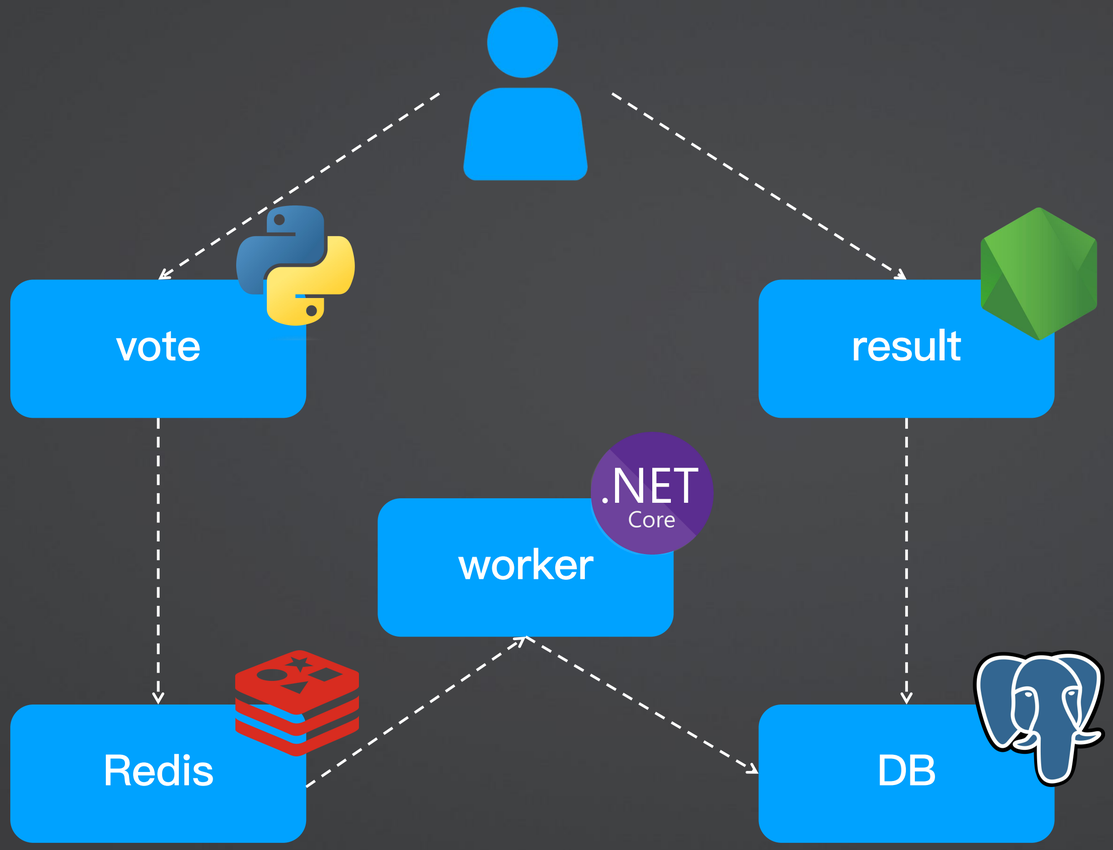
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
```

YAML 常用工具

- He3 (http://he3.app, YAML Viewer：https://t.he3app.com?06e4)
- VSCode YAML Plugin (https://marketplace.visualstudio.com/items?itemName=redhat.vscode-yaml)
- yq (https://github.com/mikefarah/yq)
  - yq 'spec.containers[0].env[0].value' pod.yaml # 读取值
  - yq '.metadata.name = "learnk8s"' pod.yaml # 写入值
  - yq '. *+ load("proxy.yaml")' app.yaml # 合并两个 YAML 文件

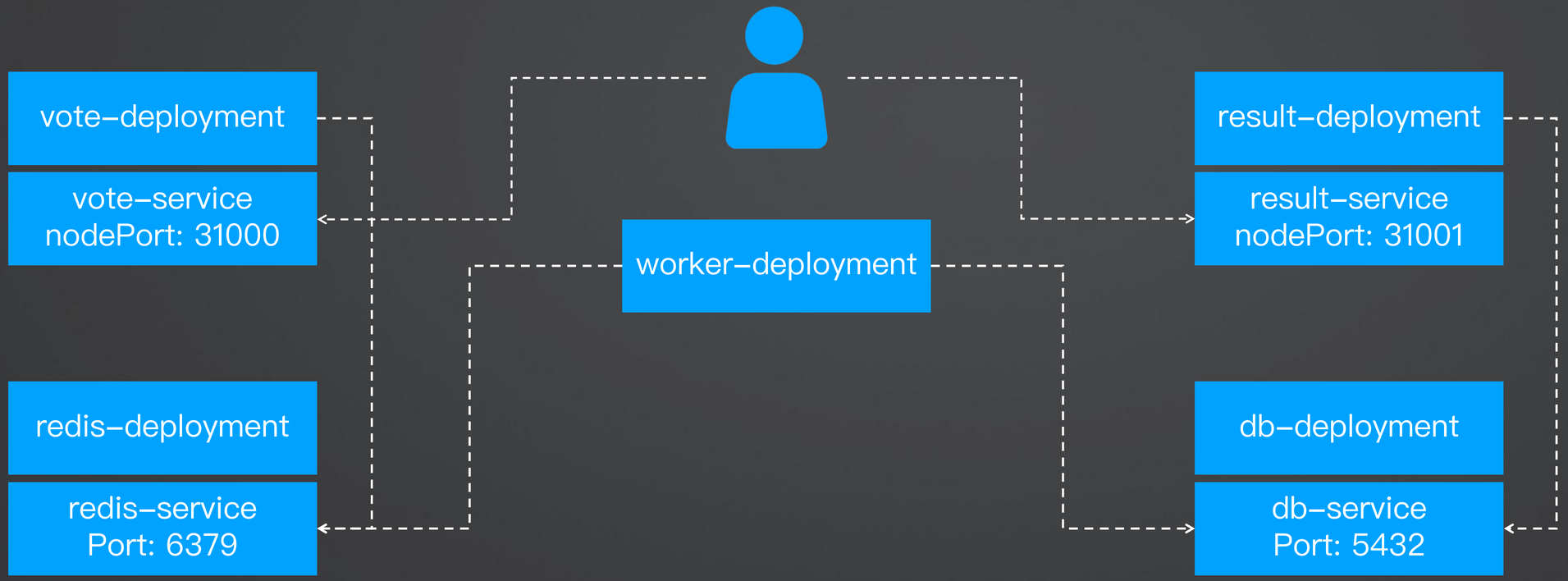
3. 微服务示例应用的设计和实现

voting-app 示例应用



Manifest

- module_3/demo_4/demo_app



vote 核心代码

极客时间 训练营

```
app = Flask(__name__)
@app.route("/", methods=['POST','GET'])
def hello():
  if request.method == 'POST':
        redis = get_redis()
        vote = request.form['vote']
        app.logger.info('Received vote for %s', vote)
        data = json.dumps({'voter_id': voter_id, 'vote': vote})
        redis.rpush('votes', data)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80, debug=True, threaded=True)
```

worker 核心代码

极客时间 训练营

```
try
{
        while (true)
        {
                Thread.Sleep(100);        # 100ms 从 redis 列表 Pop 数据
                string json = redis.ListLeftPopAsync("votes").Result;
                if (json != null)
                {
                        UpdateVote(pgsql, vote.voter_id, vote.vote);
                }
        }
}
```

```
private static void UpdateVote(NpgsqlConnection connection, string
voterId, string vote)
{
    try
    {
        command.CommandText = "INSERT INTO votes (id, vote)
VALUES (@id, @vote)";
        command.Parameters.AddWithValue("@id", voterId);
        command.Parameters.AddWithValue("@vote", vote);
        command.ExecuteNonQuery();
    }
}
```

result 核心代码

极客时间 训练营

```
async.retry(
  {times: 1000, interval: 1000},
  function(err, client) {
    if (err) {
      return console.error("Giving up");
    }
    console.log("Connected to db");
    getVotes(client);
  }
);
```

```
function getVotes(client) {
  client.query('SELECT vote, COUNT(id) AS count FROM votes GROUP BY vote', [], function(err, result) {
    if (err) {
      console.error("Error performing query: " + err);
    } else {
      var votes = collectVotesFromResult(result);
      io.sockets.emit("scores", JSON.stringify(votes));
    }
    setTimeout(function() {getVotes(client) }, 1000);
  });
}
```

result Dockerfile

```
# PID=1 将导致 nodejs 产生僵尸进程，并无法处理退出时的相关代码
process.on("SIGTERM", function onSigterm() {
  // do the cleaning job, but it wouldn't
  process.exit(0);
});
```

```
FROM node:18-slim
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
    curl \
    tini \
    && rm -rf /var/lib/apt/lists/*
WORKDIR /app
COPY package*.json ./
.....
ENTRYPOINT ["/usr/bin/tini", "--"]
CMD ["node", "server.js"]
```

https://github.com/krallin/tini

https://github.com/nodejs/docker-node/blob/main/docs/BestPractices.md#handling-kernel-signals

中间件

- Redis
- PostgreSQL
- Demo 环境以 Deployment + emptyDir 方式部署

服务间依赖问题如何解决?

极客时间 训练营

- result 服务依赖于 postgres
- worker 服务依赖于 redis
- vote 服务虽然依赖于 redis，但仍能正常启动，因为当有请求时才需连接 redis

业务重试

极客时间 训练营

```
# 业务做重试
async.retry(
  { times: 1000, interval: 1000 },
  function (callback) {
    pool.connect(function (err, client, done) {
      if (err) {
        console.error("Waiting for db");
      }
      callback(err, client);
    });
  });
```

K8s 自动重启机制

1. 由于所依赖的服务未 ready，业务进程会退出，状态码非 0
2. K8s 检测到容器异常退出，自动重启
3. 所依赖的服务仍未 ready，继续重启.....
4. 直到依赖的服务 ready，业务启动完成

K8s 自动重启机制的缺点

- K8s 重启时间采用指数退避策略
- 即下一次启动时间是上一次的 2 倍，导致应用整体启动时间变长

控制 Pod 启动顺序

```
kind: Deployment
metadata:
  name: result
  ......
initContainers:
  – name: wait–for–db
     image: ghcr.io/groundnuty/k8s–wait–for:v1.6
     imagePullPolicy: Always
     args:
       – "pod"
        – "–lapp=db"
```

https://github.com/groundnuty/k8s-wait-for

Demo：微服务启动顺序控制

极客时间 训练营

- redis –> postgres –> worker –> vote –> result

数据库表/数据如何初始化

- 业务代码初始化（ORM、脚本等）
- K8s Job 初始化
  - 通过 clone sql schema git repository，然后执行 SQL 初始化数据库

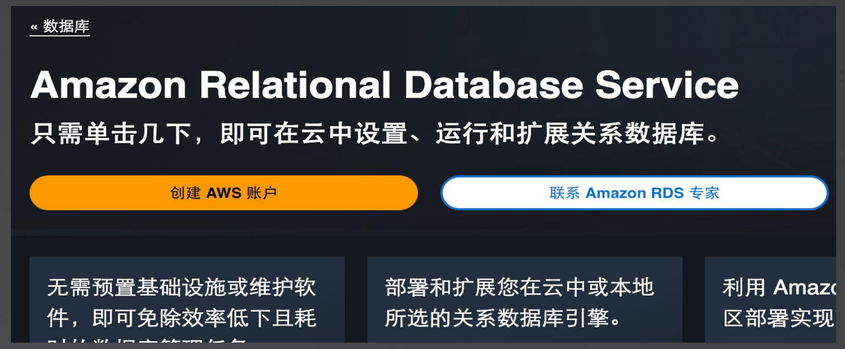
中间件高可用部署原则

极客时间 训练营

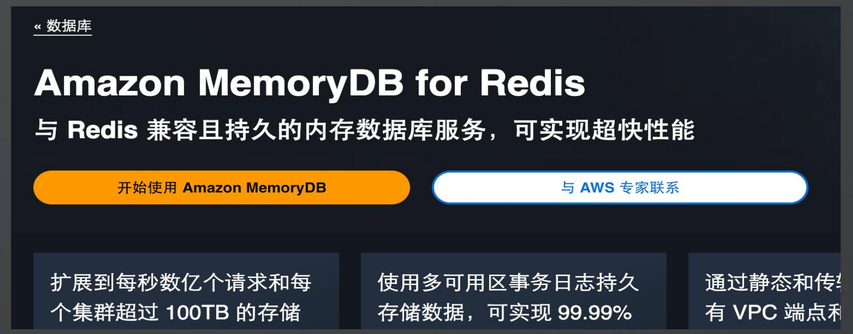
- 最佳实践：使用社区提供的 Helm Chart
  - PG: https://github.com/bitnami/charts/tree/main/bitnami/postgresql-ha
    - helm install db oci://registry-1.docker.io/bitnamicharts/postgresql-ha --set fullnameOverride=db -n db --create-namespace
  - Redis: https://github.com/bitnami/charts/blob/main/bitnami/redis/README.md
    - helm install redis oci://registry-1.docker.io/bitnamicharts/redis -n redis --create-namespace
- 生产建议：中间件尽量使用云服务，自托管次之

尽量使用云服务


云数据库 TencentDB for PostgreSQL
腾讯云数据库 PostgreSQL（TencentDB for PostgreSQL，云 API 使用 postgres 作为简称）能够让您在云端轻松设置、操作和扩展目前功能最强大的开源数据库 PostgreSQL。腾讯云将负责绝大部分处理复杂而耗时的管理工作，如 PostgreSQL 软件安装、存储管理、高可用复制、以及为灾难恢复而进行的数据备份，让您更专注于业务程序开发。
立即选购
产品文档
管理控制台


« 数据库
Amazon Relational Database Service
只需单击几下，即可在云中设置、运行和扩展关系数据库。
创建 AWS 账户
联系 Amazon RDS 专家
无需预置基础设施或维护软件，即可免除效率低下且耗
部署和扩展您在云中或本地所选的关系数据库引擎。
利用 Amaz
区部署实现


云数据库 TencentDB for Redis
观看产品
腾讯云数据库 Redis（TencentDB for Redis）是腾讯云打造的兼容 Redis 协议的缓存和存储服务。丰富的数据结构能帮助您完成不同类型的业务场景开发。支持主从热备，提供自动容灾切换、数据备份、故障迁移、实例监控、在线扩容、数据回档等全套的数据库服务。
立即选购
产品文档


« 数据库
Amazon MemoryDB for Redis
与 Redis 兼容且持久的内存数据库服务，可实现超快性能
开始使用 Amazon MemoryDB
与 AWS 专家联系
扩展到每秒数亿个请求和每个集群超过 100TB 的存储
使用多可用区事务日志持久存储数据，可实现 99.99%
通过静态和传
有 VPC 端点

4. 应用定义：Helm 入门和实战

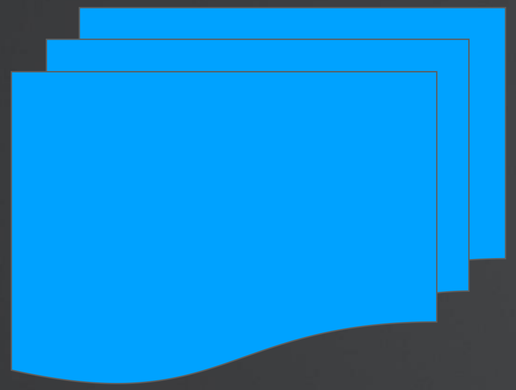
K8s 微服务

使用 Helm 定义应用

极客时间 训练营

- 管理 K8s 对象
- 将多个微服务的工作负载、配置对象等封装为一个应用
- 屏蔽终端用户使用的复杂度
- 参数化、模板化，支持多环境

Helm 核心概念



Chart

K8s 应用安装包，包含应用的 K8s 对象
用于创建实例

Release

使用默认或特定参数安装的 Helm 实例
（运行中的实例）

Repository

用于存储和分发的 Helm Chart 仓库
（Git、OCI）

Helm 安装

- curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
- https://helm.sh/docs/intro/install/

Helm 常用命令

极客时间 训练营

- **install**：安装 Helm Chart
- **get, status, list**：获取 Helm Release 的信息，例如安装状态、日志、安装参数等
- **uninstall**：卸载 Helm Release
- **repo add, repo list, repo remove, repo index**： Repository 相关命令
- **search**：在 Repository 中查找 Helm Chart
- **create, package**：创建和打包 Helm Chart
- **pull**：拉取 helm chart

Demo: 从零创建 Helm Chart

极客时间 训练营

- helm create demo
- 核心目录和文件
  - templates 目录
    - 存储模板文件
  - Chart.yaml
    - Helm Chart metadata
  - values.yaml
    - 模板变量

Chart.yaml

极客时间 训练营

```
# 红字部分是必须，其他非必须


apiVersion: v2    # API 版本，默认v2
name: demo    # Helm Chart 名称
description: A Helm chart for Kubernetes    # 描述
type: application    # application 或 library，对于应用选择前者
version: 0.1.0    # Helm Chart 版本号
appVersion: "1.16.0"     # 应用版本号
dependencies:            # 依赖的其他 chart
......
```

values.yaml 和模板变量

```
# 所有字段和值都可自定义，用来为模板提供值
replicaCount: 1
image:
  repository: nginx
  pullPolicy: IfNotPresent
  # Overrides the image tag whose default is the chart appVersion.
  tag: "v1.0"
imagePullSecrets: []
nameOverride: ""
fullnameOverride: ""
......
```

```
# templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag}}"
                    # 该模板变量替换为 values.yaml 配置的值
```

Demo: 将示例应用修改为 Helm

- 要抽哪些变量和模板?
  - Secret、ConfigMap 等配置文件
- 如何调试?
  - helm template .
  - helm inspect values oci://registry-1.docker.io/bitnamicharts/redis
- 多环境问题如何处理?
  - 使用不同的 values.yaml 文件
- 生产环境下如何控制不部署中间件?
  - 使用 conditions

Helm 高级核心技术

极客时间 训练营

- 内置变量和函数
  - https://helm.sh/docs/chart_template_guide/builtin_objects/
  - https://helm.sh/docs/chart_template_guide/function_list
- 依赖
  - https://helm.sh/docs/helm/helm_dependency
- 渲染和调试
  - helm template . --debug
- Helm Hooks
  - https://helm.sh/docs/topics/charts_hooks/
  - pre-install、post-install、pre-delete、post-delete、pre-upgrade、post-upgrade、pre-rollback、post-rollback

使用第三方包

- GitHub
  - https://github.com/bitnami/charts
- Artifact Hub
  - https://artifacthub.io/

依赖（dependency）


```yaml
dependencies:
  - name: redis
    version: "17.16.0"
    repository: "oci://registry-1.docker.io/bitnamicharts"
    condition: redis.enabled
    tags:
      - middleware
  - name: postgresql-ha
    version: "11.9.0"
    repository: "oci://registry-1.docker.io/bitnamicharts"
    condition: postgresql-ha.enabled
    tags:
      - middleware
```

依赖（Dependencies）

极客时间 训练营

```
# Chart.yaml

dependencies:

– name: nginx

  version: "1.2.3"

  repository: "https://example.com/charts"

– name: memcached

  version: "1.2.3"

  repository: "file://../dependency_chart/memcached"
```

内置变量

- Release
  - Release.Name
  - Release.Namespace
  - Release.IsUpgrade
  - Release.IsInstall
  - Release.Revision
  - Release.Service
- Values
  - {{ .Values.Obj }}
- Chart
  - {{ .Chart.Name }}
- Files
  - Files.Get
  - Files.Glob
  - Files.AsSecrets
  - Files.AsConfig
- Capabilities
  - Capabilities.APIVersions
  - Capabilities.KubeVersion
- Template
  - Template.Name
  - Template.BasePath

内置函数

- Date
- Encoding
- File Path
- Lists
- Math
- Network
- Regular Expressions
- String
- Type Conversion
- URL
- UUID

子 Chart（Sub Chart）

Sub Chart 和 dependencies 依赖的关系

- dependencies 是借助子 chart 来实现的
- 子 chart 约定放在 charts 目录，和 templates 目录同级
- helm 会将依赖下载到 charts 目录
- 子 chart 也可以自己创建
- 父 chart 可以覆写子 chart 的 values.yaml 值，但子 chart 不能覆写父 chart values.yaml 值

  subchart:

    key: values

- values 有一个保留关键字：global，子和父 chart 都能访问

  global:

    salad: caesar

Hooks

- pre-install：渲染模板之后、创建任何资源之前执行 (helm install)
- post-install：所有资源部署到 Kubernetes 后执行 (helm install)
- pre-delete：在从 Kubernetes 中删除任何资源之前执行 (helm uninstall)
- post-delete：在删除所有资源后执行 (helm uninstall)
- pre-upgrade：在渲染模板后、更新资源前执行 (helm upgrade)
- post-upgrade：在升级所有资源后执行 (helm upgrade)
- pre-rollback：在渲染模板之后、回滚任何资源之前执行 (helm rollback)
- post-rollback：在修改所有资源后执行 (helm rollback)
- test：在执行 helm test 命令时执行

Demo：Hooks

```
apiVersion: batch/v1
kind: Job
metadata:
  name: "{{ .Release.Name }}"
  annotations:
    # This is what defines this resource as a hook. Without this line, the
    # job is considered part of the release.
    "helm.sh/hook": post-install
    "helm.sh/hook-weight": "-5"
    "helm.sh/hook-delete-policy": hook-succeeded
```

Helm Cheat Sheet

极客时间 训练营

```
# 安装 Helm Chart 或者执行升级，无需担心重复执行问题

$ helm upgrade --install <release name> --values <values file> <chart directory>

# 查看 Helm Chart values.yaml 配置信息

$ helm inspect values <CHART>

# 查看 Helm Repository Chart 列表

$ helm repo list

# 查看所有命名空间的 release

$ helm list --all-namespaces

# 升级应用前先更新依赖

$ helm upgrade <release> <chart> --dependency-update

# 回滚应用

$ helm rollback <release> <revision>
```

6. 应用定义：Kustomize 入门和实战

什么是 Kustomize

- Kustomize 是一个 CLI 工具
- 可以对 Manifest 的任何字段进行覆写
- 尤其适合多环境的场景
- 由 Kubernetes 团队开发，并已经内置到 kubectl 工具

例子

极客时间 训练营

```
├── base                    # 基础目录，或者不同环境的通用 Manifest
│   ├── deployment.yaml
│   ├── kustomization.yaml
└── overlays
    ├── dev                 # dev 环境目录
    │   ├── hpa.yaml
    │   └── kustomization.yaml
    ├── production          # production 环境目录
    │   ├── hpa.yaml
    │   ├── kustomization.yaml
    └── staging             # staging 环境目录
        ├── hpa.yaml
        ├── kustomization.yaml
```

kustomization.yaml 常用配置

极客时间 训练营

1. resource：定义 Manifest 资源，文件、目录或 URL
2. secretGenerator：生成 secret 对象
3. configMapGenerator：生成 configMap 对象
4. images：覆写 image tag
5. helmCharts：定义依赖的 helm chart
6. patchesStrategicMerge：覆写操作，v1 版本将废弃，建议迁移到 patches

更多配置：https://kubectl.docs.kubernetes.io/references/kustomize/kustomization/

Demo: 将示例应用修改为 Kustomize

1. 不含 Overlays 目录最小化的 Kustomize 应用定义
2. 增加 overlays/dev 目录，并提取镜像、覆写 DB 和 Redis 连接信息、覆写 OPTION
3. 增加中间件部署（Kustomize 和 Helm Chart 混用）

部署：kubectl kustomize ./overlays/dev | kubectl apply –f –

部署（含 Helm Chart 依赖）：kubectl kustomize ./overlays/dev ––enable–helm | kubectl apply –f –

有了 Helm 为什么还需要 Kustomize?

极客时间 训练营

1. 可以覆写任何 Manifest 字段和值
    1. 例如增加 Annotations
    2. 修改 Labels
2. 编写简单，无需学习复杂的模板变量语法
3. 多环境下、Base 和 Overlays 模式能够实现很好的 Manifest 复用
4. Helm 隐藏应用细节，对最终用户友好，Kustomize 暴露所有 K8s API，对开发者友好

缺点

1. 分发没有 Helm Chart 方便
2. 生态相比较 Helm 差
3. 强依赖于目录结构

THANKS