

无限可能 享受不同滋味

按自己的方式去拼



一些 CI 的心得

老麦



微信搜一搜



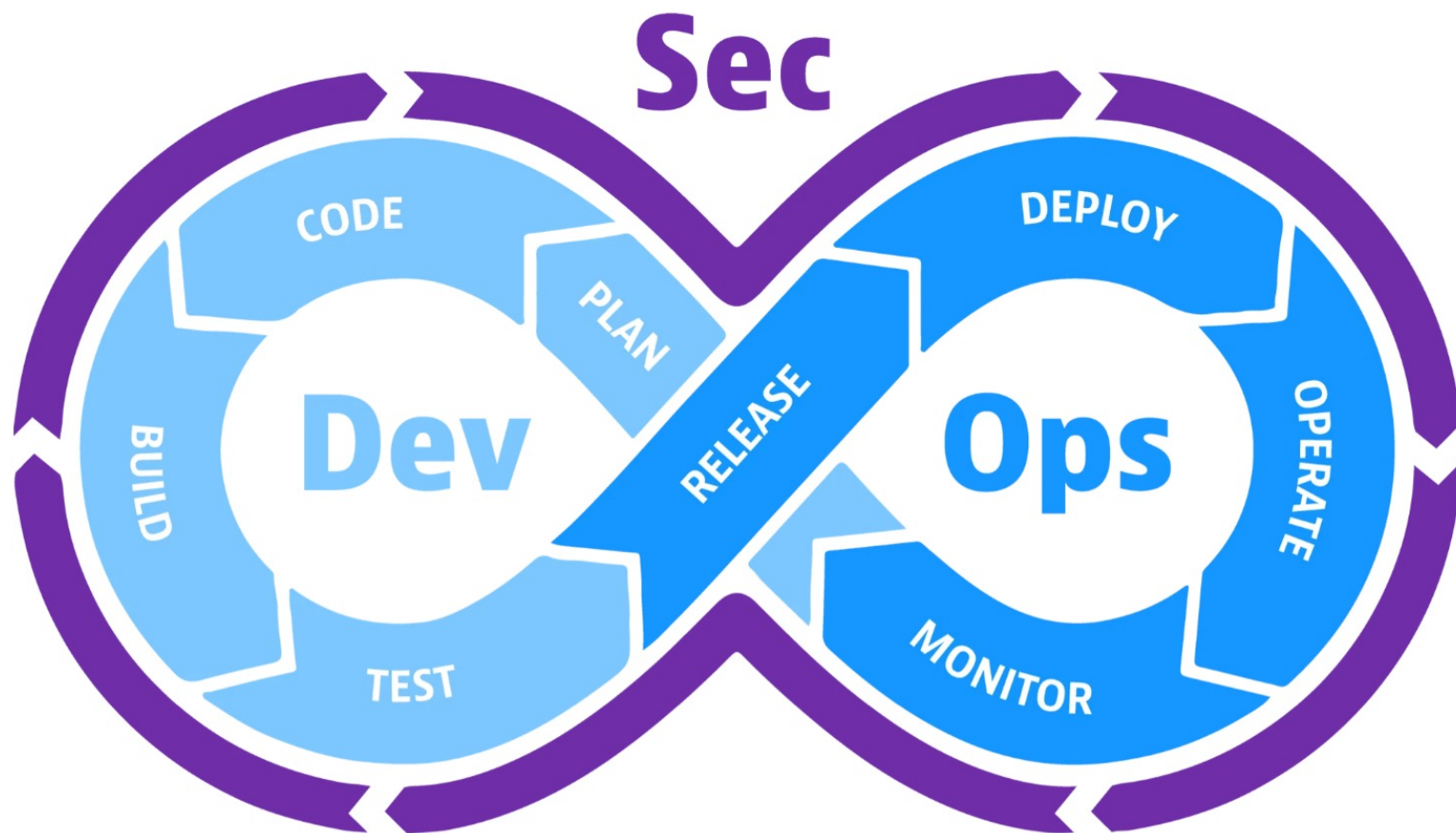
熊猫云原生Go

目录

1.CI 工具

2. 项目容器化

CI / CD 的定义



1. “CI”始终指**持续集成**，它属于开发人员的自动化流程。成功的 CI 表明应用代码的新更改会定期构建、测试并合并到共享存储库中。这种方法可以解决在一次开发中有太多应用分支，从而**导致相互冲突的问题**。
2. “CD”指的是**持续交付**和/或**持续部署**，这些相关概念有时会交叉使用。两者都事关管道后续阶段的自动化，但它们有时也会单独使用，用于说明自动化程度。

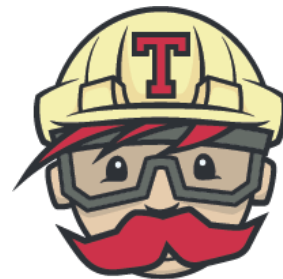
流水线



CI 工具的选择



Jenkins



Travis CI



GitLab

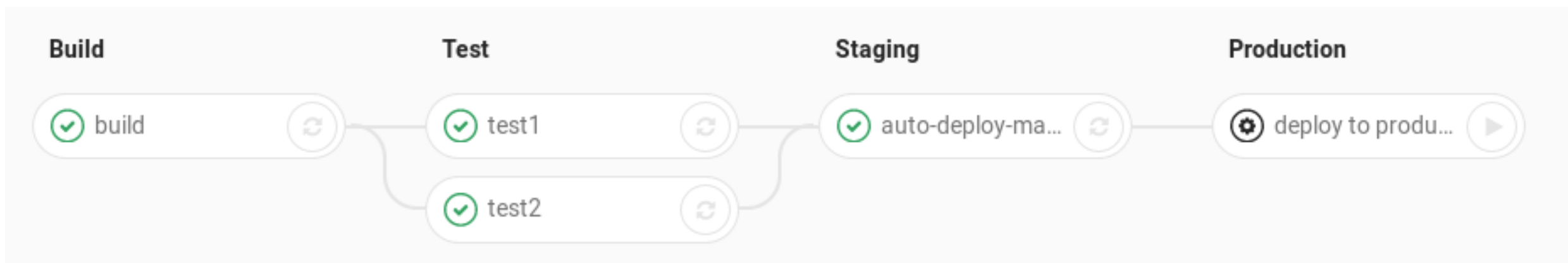


circleci

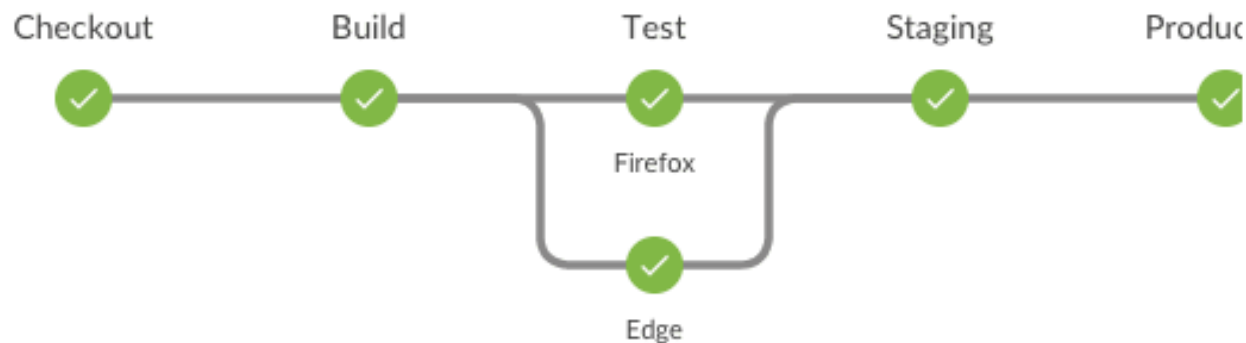


DRONE
by harness

殊途同归



Jenkins Pipeline



Shell: 最通用的任务实现

执行方式

1. 命令行
2. API 调用 => cURL ==> 最终还是命令行

所以， 最终的执行方式， 还是 命令行 => bash / sh 环境

调用工具 => 开发工具 => **执行工具** <= 配置执行环境

.travis.yml

```
os: osx

before_install:
- brew install git-lfs

before_script:
- git lfs pull
```

.gitlab-ci.yml 文件可能包含:

```
stages:
- build
- test

build-code-job:
  stage: build
  script:
    - echo "Check the ruby version,"
    - ruby -v
    - rake
```

github.action.yml

```
34 - name: Build Binary
35   run: make build.x
36
37 - uses: "marvinpinto/action-automatic-releases@latest"
38   with:
39     repo_token: "${{ secrets.GITHUB_TOKEN }}"
40     prerelease: false
41     files: |
42       out/*
```

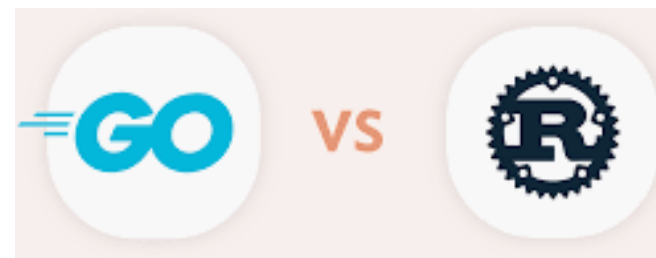

流水线的两部分

1. 流水线的「阶段」规划



2. 流水线的「任务」如何完成?

自己实现



管道 - 最常见的 CI

```
→ ls -al | awk '{print $1}' | sort | uniq -c | sort -r  
29 drwxr-xr-x  
24 -rw-r--r--  
10 lrwxr-xr-x  
6 drwx-----  
6 -rw-----  
4 drwx-----@  
4 drwx-----+
```

还是说回来： 自己的工具实现

```
→ ax docker -h  
generate dockerfile, do docker building
```

Usage:

```
ax docker [command]
```

Available Commands:

build	docker build
buildx	docker buildx build
file	generate dockerfiles
image-name	return docker image name
kustomize	generate kustomization configs
runtime-image-name	return docker image name
transmit	transmit images from container registry A to B

Flags:

```
-h, --help    help for docker
```

ax = cat = sed = awk = docker

执行环境： 宿主机和容器

1. 本质上没有区别。

2. 同时也没有银弹。

宿主机：

- 优势： 简单，
- 劣势： 但并行任务中有环境冲突。

容器：

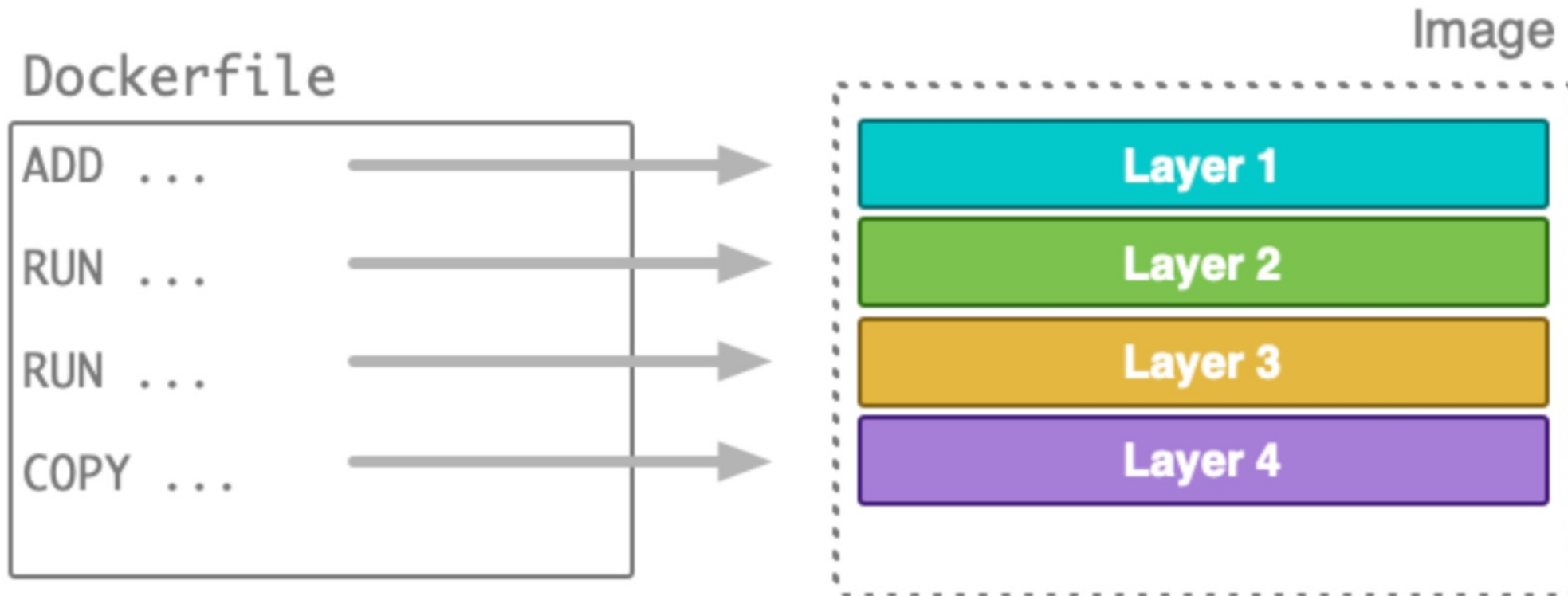
- 优势： 环境独立。
- 劣势： 配置不难， 但是资源管理比较绕。

目录

1. CI 工具

2. 项目容器化

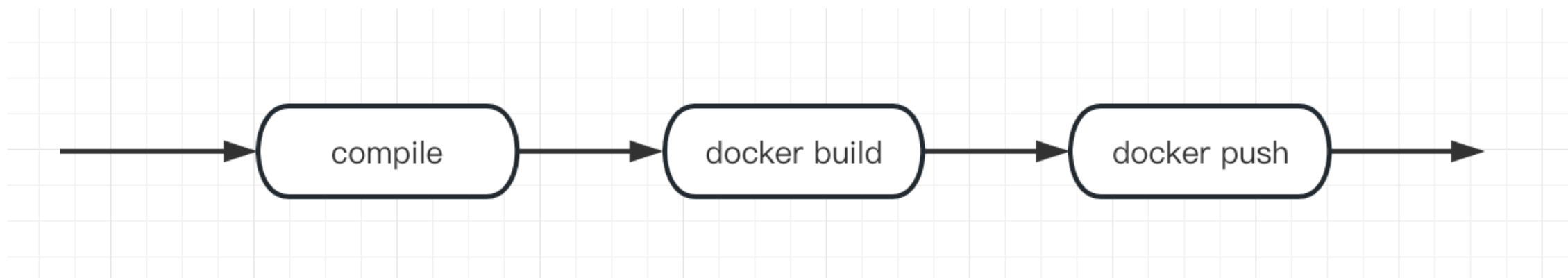
项目容器化: Dockerfile 合理分层



Dockerfile 优化 (1) 分层: <https://mp.weixin.qq.com/s/pen6G2aFsPfSghKjgocVjQ>

Dockerfile 优化 (2) ARG: <https://mp.weixin.qq.com/s/i3n0hoHRaYoDMWcC5DSQ3w>

常规操作

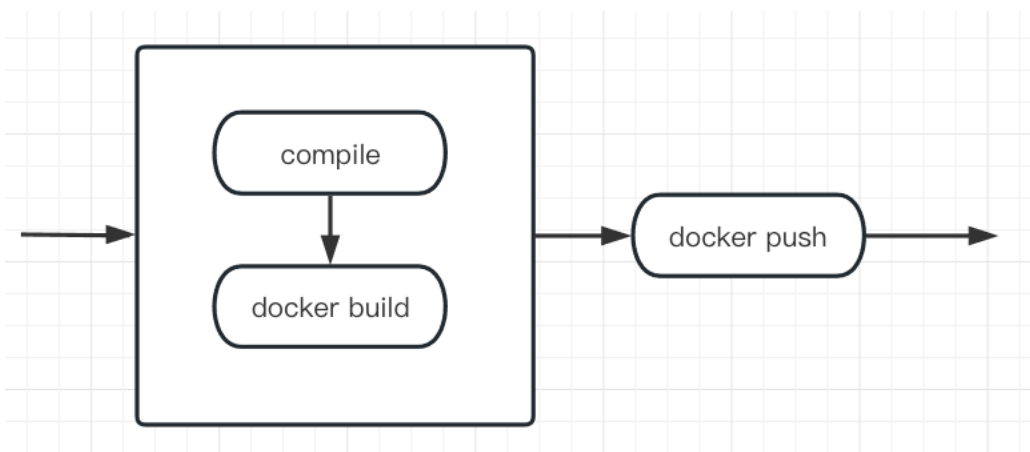


为了更好的优化镜像分层， 我们可以怎么做？

多阶段构建

常见于「编译型」语言：

1. Go
2. Java

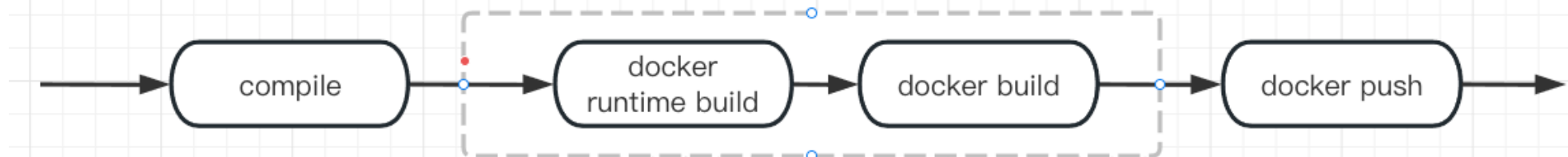


```
1  
2 # go compile  
3 FROM golang:1.19 as builder  
4 WORKDIR /app  
5 COPY . .  
6 RUN GGO_ENABLED=0 go build -o demo .  
7  
8 # docker image build  
9 FROM alpine:3  
10 WORKDIR /app  
11 COPY --from=builder /app/demo /app/demo  
12 ENTRYPOINT ["/app/demo"]  
13
```

多任务构建 (1)

常见于「解释性」语言：

1. Python
2. Ruby
3. Ndoejs




```
2 # Dockerfile.runtime
3 ## docker build -t example.com/project/repo:runtime-hash .
4 FROM python:3 as runtime
5 WORKDIR app
6 COPY requirement.txt .
7 RUN pip install -r requirement.txt
8
```

```
10 # Dockerfile
11 ## docker build -t example.com/project/repo:version \
12 ## . . . --build-arg=RUNTIME_IMAGE=example.com/project/repo:runtime-hash .
13 ARG RUNTIME_IMAGE
14 FROM ${RUNTIME_IMAGE}
15 COPY . .
16 ENTRYPOINT ["python3", "start.py"]
17
```

多任务构建（2）

```
2 # Dockerfile.runtime
3 ## docker build -t example.com/project/repo:runtime-hash .
4 FROM python:3 as runtime
5 WORKDIR app
6 COPY requirement.txt .
7 RUN pip install -r requirement.txt
```



关于 Hash 的计算： 规则自己定

1. **Dockerfile, Dockerfile.runtime** 是公共的
2. Python: **requirement.txt** ...
3. Nodejs: **package.json, yarn.lock** ...
4. Ruby: **Gemfile, Gemfile.lock** ...

Md5 => Dockerfile +dockerfile.runtime + req.txt
=> 123asdfjalskdfjasdlfjalsdjfl+xxx+xxxx
Md5 => xcvk320dfadf

谢谢 & QA



微信搜一搜



熊猫云原生Go