关于Dockerfile的最佳实践技巧

Dockerfile的语法非常简单，然而如何加快镜像构建速度，如何减少Docker镜像的大小却不是那么直观，需要积累实践经验。这篇文章可以帮助你快速掌握编写Dockerfile的技巧。

- 更快的构建速度
- 更小的Docker镜像大小
- 更少的Docker镜像层
- 充分利用镜像缓存
- 增加Dockerfile可读性
- 让Docker容器使用起来更简单
- 编写.dockerignore文件
- 容器只运行单个应用
- 将多个RUN指令合并为一个
- 基础镜像的标签不要用latest
- 每个RUN指令后删除多余文件
- 选择合适的基础镜像(alpine版本最好)
- 设置WORKDIR和CMD
- 使用ENTRYPOINT (可选)
- 在entrypoint脚本中使用exec
- COPY与ADD优先使用前者
- 合理调整COPY与RUN的顺序
- 设置默认的环境变量，映射端口和数据卷
- 使用LABEL设置镜像元数据
- 添加HEALTHCHECK
- 多阶段构建

示例Dockerfile犯了几乎所有的错(当然我是故意的)。接下来，我会一步步优化它。假设我们需要使用Docker运行一个Node.js应用，下面就是它的Dockerfile(CMD指令太复杂了，所以我简化了，它是错误的，仅供参考)。

```
FROM ubuntu
ADD . /app
RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y nodejs ssh mysql
RUN cd /app && npm install
# this should start three processes, mysql and ssh
# in the background and node app in foreground
```

```
# isn't it beautifully terrible? <3
CMD mysql & sshd & npm start
```

构建镜像:

```
docker build -t wtf .
```

你能发现上面Dockerfile所有的错误吗? 不能? 那接下来让我们一步一步完善它。

1. 编写.dockerignore文件

构建镜像时，Docker需要先准备 `context` ，将所有需要的文件收集到进程中。默认的 `context` 包含Dockerfile目录中的所有文件，但是实际上，**我们并不需要.git目录，node_modules目录等内容。** `.dockerignore` 的作用和语法类似于 `.gitignore` ，可以忽略一些不需要的文件，这样可以有效加快镜像构建时间，同时减少Docker镜像的大小。示例如下:

```
.git/node_modules/
```

2. 容器只运行单个应用

从技术角度讲，你可以在Docker容器中运行多个进程。你可以将数据库，前端，后端，ssh，supervisor都运行在同一个Docker容器中。但是，这会让你非常痛苦:

- 非常长的构建时间(修改前端之后，整个后端也需要重新构建)
- 非常大的镜像大小
- 多个应用的日志难以处理(不能直接使用stdout，否则多个应用的日志会混合到一起)
- 横向扩展时非常浪费资源(不同的应用需要运行的容器数并不相同)
- 僵尸进程问题 - 你需要选择合适的init进程

因此，建议大家为每个应用构建单独的Docker镜像，然后使用 Docker Compose 运行多个Docker容器。
现在，我从Dockerfile中删除一些不需要的安装包，另外，SSH可以用docker exec替代。示例如下:

```
FROM ubuntu
ADD . /app
RUN apt-get update
RUN apt-get upgrade -y
```

```
# we should remove ssh and mysql, and use
# separate container for database
RUN apt-get install -y nodejs    # ssh mysql
RUN cd /app && npm install
CMD npm start
```

3. 将多个RUN指令合并为一个

Docker镜像是分层的，下面这些知识点非常重要:

- Dockerfile中的每个指令都会创建一个新的镜像层。
- 镜像层将被缓存和复用
- 当Dockerfile的指令修改了，复制的文件变化了，或者构建镜像时指定的变量不同了，对应的镜像层缓存就会失效
- 某一层的镜像缓存失效之后，它之后的镜像层缓存都会失效
- 镜像层是不可变的，如果我们再某一层中添加一个文件，然后在下一层中删除它，则镜像中依然会包含该文件(只是这个文件在Docker容器中不可见了)。

Docker镜像类似于洋葱。它们都有很多层。为了修改内层，则需要将外面的层都删掉。记住这一点的话，其他内容就很好理解了。
现在，我们**将所有的RUN指令合并为一个**。同时把 `apt-get upgrade` 删除，因为它会使得镜像构建非常不确定(我们只需要依赖基础镜像的更新就好了)

```
FROM ubuntu
ADD . /app
RUN apt-get update \
      && apt-get install -y nodejs \
      && cd /app \
      && npm install
CMD npm start
```

记住一点，我们只能将变化频率一样的指令合并在一起。将node.js安装与npm模块安装放在一起的话，则每次修改源代码，都需要重新安装node.js，这显然不合适。因此，正确的写法是这样的:

```
FROM ubuntu
RUN apt-get update && apt-get install -y nodejs
ADD . /app
```

```
RUN cd /app && npm install
CMD npm start
```

4. 基础镜像的标签不要用latest

当镜像没有指定标签时，将默认使用 `latest` 标签。因此， `FROM ubuntu` 指令等同于 `FROM ubuntu:latest` 。当时，当镜像更新时，latest标签会指向不同的镜像，这时构建镜像有可能失败。如果你的确需要使用最新版的基础镜像，可以使用latest标签，否则的话，最好指定确定的镜像标签。
示例Dockerfile应该使用 `16.04` 作为标签。

```
FROM ubuntu:16.04   # it's that easy!
RUN apt-get update && apt-get install -y nodejs
ADD . /app
RUN cd /app && npm install
CMD npm start
```

5. 每个RUN指令后删除多余文件

假设我们更新了apt-get源，下载，解压并安装了一些软件包，它们都保存
在 `/var/lib/apt/lists/` 目录中。但是，运行应用时Docker镜像中并不需要这些文件。我们最好将它们删除，因为它会使Docker镜像变大。
示例Dockerfile中，我们可以删除 `/var/lib/apt/lists/` 目录中的文件(它们是由apt-get update生成的)。

```
FROM ubuntu:16.04
RUN apt-get update \
      && apt-get install -y nodejs \
      # added lines
      && rm -rf /var/lib/apt/lists/*
ADD . /app
RUN cd /app && npm install
CMD npm start
```

6. 选择合适的基础镜像(alpine版本最好)

在示例中，我们选择了 ubuntu 作为基础镜像。但是我们只需要运行node程序，有必要使用一个通用的基础镜像吗？ node 镜像应该是更好的选择。

```
FROM node
ADD . /app
# we don't need to install node
# anymore and use apt-get
RUN cd /app && npm install
CMD npm start
```

更好的选择是alpine版本的 node 镜像。alpine是一个极小化的Linux发行版，只有4MB，这让它非常适合作为基础镜像。

```
FROM node:7-alpine
ADD . /app
RUN cd /app && npm install
CMD npm start
```

apk是Alpine的包管理工具。它与 apt-get 有些不同，但是非常容易上手。另外，它还有一些非常有用的特性，比如 no-cache 和 --virtual 选项，它们都可以帮助我们减少镜像的大小。

7. 设置WORKDIR和 CMD

WORKDIR指令可以设置默认目录，也就是运行 RUN / CMD / ENTRYPOINT 指令的地方。
CMD指令可以设置容器创建是执行的默认命令。另外，你应该讲命令写在一个数组中，数组中每个元素为命令的每个单词(参考官方文档)。

```
FROM node:7-alpine
WORKDIR /app
ADD . /app
RUN npm install
CMD ["npm", "start"]
```

8. 使用ENTRYPOINT (可选)

ENTRYPOINT指令并不是必须的，因为它会增加复杂度。 `ENTRYPOINT` 是一个脚本，它会默认执行，并且将指定的命令当成参数接收。它通常用于构建可执行的Docker镜像。entrypoint.sh如下:

```
#!/usr/bin/env sh_# $0 is a script name, # 2, $3 etc are passed arguments# 1case "$CMD" in "dev" ) npm install export NODE_ENV=development exec npm run dev ;; "start" ) _# we can modify files here, using ENV variables passed in _ # "docker create" command. It can't be done during build process. echo "db: $DATABASE_ADDRESS" >> /app/config.yml export NODE_ENV=production exec npm start ;; * ) _# Run custom command. Thanks to this line we can still use _ # "docker run our_image /bin/bash" and it will work exec {@:2} ;;esac
```

示例Dockerfile:

```
FROM node:7-alpine
WORKDIR /app
ADD . /app
RUN npm install
ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

可以使用如下命令运行该镜像:

```
_# 运行开发版本_docker run our-app dev _# 运行生产版本_docker run our-app start _# 运行bash_docker run -it our-app /bin/bash
```

9. 在entrypoint脚本中使用exec

在前文的entrypoint脚本中，我使用了 `exec` 命令运行node应用。不使用 `exec` 的话，我们则不能顺利地关闭容器，因为SIGTERM信号会被bash脚本进程吞没。 `exec` 命令启动的进程可以取代脚本进程，因此所有的信号都会正常工作。
这里扩展介绍一下docker容器的停止过程:
(1). 对于容器来说， `init` 系统不是必须的，当你通过命令 `docker stop mycontainer` 来停止容器时，docker CLI 会将 `TERM` 信号发送给 mycontainer 的 `PID` 为 1 的进程。

- **如果 PID 1 是 init 进程** - 那么 PID 1 会将 TERM 信号转发给子进程，然后子进程开始关闭，最后容器终止。

- **如果没有 init 进程**- 那么容器中的应用进程（Dockerfile 中的 `ENTRYPOINT` 或 `CMD` 指定的应用）就是 PID 1，应用进程直接负责响应 `TERM` 信号。这时又分为两种情况：
  - **应用不处理 SIGTERM** - 如果应用没有监听 `SIGTERM` 信号，或者应用中没有实现处理 `SIGTERM` 信号的逻辑，应用就不会停止，容器也不会终止。
  - **容器停止时间很长** - 运行命令 `docker stop mycontainer` 之后，Docker 会等待 `10s`，如果 `10s` 后容器还没有终止，Docker 就会绕过容器应用直接向内核发送 `SIGKILL`，内核会强行杀死应用，从而终止容器。

(2).如果容器中的进程没有收到 `SIGTERM` 信号，很有可能是因为应用进程不是 `PID 1`，PID 1 是 `shell`，而应用进程只是 `shell` 的子进程。而 shell 不具备 `init` 系统的功能，也就不会将操作系统的信号转发到子进程上，这也是容器中的应用没有收到 `SIGTERM` 信号的常见原因。

问题的根源就来自 `Dockerfile`，例如：

```
FROM  alpine:3.7
COPY  popcorn.sh  .
RUN  chmod  +x  popcorn.sh
ENTRYPOINT  ./popcorn.sh
CMD  ["start"]
```

`ENTRYPOINT` 指令使用的是 **shell 模式**，这样 Docker 就会把应用放到 `shell` 中运行，因此 `shell` 是 PID 1。
解决方案有以下几种：

方案 1：使用 exec 模式的 ENTRYPOINT 指令

与其使用 shell 模式，不如使用 exec 模式，例如：

```
FROM  alpine:3.7
COPY  popcorn.sh  .
RUN  chmod  +x  popcorn.sh
ENTRYPOINT  ["./popcorn.sh"]
```

这样 PID 1 就是 `./popcorn.sh`，它将负责响应所有发送到容器的信号，至于 `./popcorn.sh` 是否真的能捕捉到系统信号，那是另一回事。
举个例子，假设使用上面的 Dockerfile 来构建镜像，`popcorn.sh` 脚本每过一秒打印一次日期：

```
#!/bin/sh

while true
do
        date
        sleep 1
done
```

构建镜像并创建容器：

```
docker build -t truek8s/popcorn .
docker run -it --name corny --rm truek8s/popcorn
```

打开另外一个终端执行停止容器的命令，并计时：

```
time docker stop corny
```

因为 `popcorn.sh` 并没有实现捕获和处理 `SIGTERM` 信号的逻辑，所以需要 10s 左右才能停止容器。要想解决这个问题，就要往脚本中添加信号处理代码，让它捕获到 `SIGTERM` 信号时就终止进程：

```
#!/bin/sh
# catch the TERM signal and then exit
trap "exit" TERM
while true
do
        date
        sleep 1
done
```

注意：下面这条指令与 shell 模式的 ENTRYPOINT 指令是等效的：

```
ENTRYPOINT ["/bin/sh", "./popcorn.sh"]
```

方案 2：直接使用 exec 命令

如果你就想使用 `shell` 模式的 ENTRYPOINT 指令，也不是不可以，只需将启动命令追加到 `exec` 后面即可，例如：

```
FROM alpine:3.7
COPY popcorn.sh .
RUN chmod +x popcorn.sh
ENTRYPOINT exec ./popcorn.sh
```

这样 `exec` 就会将 shell 进程替换为 `./popcorn.sh` 进程，PID 1 仍然是 `./popcorn.sh`。

方案 3：使用 init 系统

如果容器中的应用默认无法处理 `SIGTERM` 信号，又不能修改代码，这时候方案 1 和 2 都行不通了，只能在容器中添加一个 `init` 系统。init 系统有很多种，这里推荐使用 tini，它是专用于容器的轻量级 init 系统，使用方法也很简单：

1. 安装 `tini`
2. 将 `tini` 设为容器的默认应用
3. 将 `popcorn.sh` 作为 `tini` 的参数

具体的 Dockerfile 如下：

```
FROM alpine:3.7
COPY popcorn.sh .
RUN chmod +x popcorn.sh
RUN apk add --no-cache tini
ENTRYPOINT ["/sbin/tini", "--", "./popcorn.sh"]
```

现在 ``` tini

```
  就是 PID 1，它会将收到的系统信号转发给子进程 ```
popcorn.sh
```

。

10. COPY与ADD优先使用前者

COPY指令非常简单，仅用于将文件拷贝到镜像中。ADD相对来讲复杂一些，可以用于下载远程文件以及解压压缩包(参考官方文档)。

```
FROM  node:7-alpine
WORKDIR  /app
COPY  .  /app
RUN  npm  install
ENTRYPOINT  ["./entrypoint.sh"]
CMD  ["start"]
```

11. 合理调整COPY与RUN的顺序

我们应该**把变化最少的部分放在Dockerfile的前面**，这样可以充分利用镜像缓存。
在构建镜像的时候,docker 会按照 `dockerfile` 中的指令顺序来一次执行。每一个指令被执行的时候 docker 都会去缓存中检查是否有已经存在的镜像可以复用，而不是去创建一个新的镜像复制。如果不想使用构建缓存,可以使用 `docker build` 参数选项 `--no-cache=true` 来禁用构建缓存。在使用镜像缓存时,要弄清楚缓存合适生效,何时失效。构建缓存最基本规则如下:

- 如果引用的父镜像在构建缓存中,下一个命令将会和所有从该父进程派生的子镜像做比较,如果有子镜像使用相同的命令,那么缓存命中,否则缓存失效。
- 在大部分情况下,通过比较 `Dockerfile` 中的指令和子镜像已经足够了。但是有些指令需要进一步的检查。
- 对于 `ADD` 和 `COPY` 指令, 文件的内容会被检查,并且会计算每一个文件的校验码。但是文件最近一次的修改和访问时间不在校验码的考虑范围内。在构建过程中,docker 会比对已经存在的镜像,只要有文件内容和元数据发生变动，那么缓存就会失效。
- 除了 `ADD` 和 `COPY` 指令,镜像缓存不会检查容器中文件来判断是否命中缓存。例如,在处理 `RUN apt-get -y update` 命令时,不会检查容器中的更新文件以确定是否命中缓存,这种情况下只会检查命令字符串是否相同。

示例中，源代码会经常变化，则每次构建镜像时都需要重新安装NPM模块，这显然不是我们希望看到的。因此我们可以先拷贝 `package.json` ，然后安装NPM模块，最后才拷贝其余的源代码。这样的话，即使源代码变化，也不需要重新安装NPM模块。

```
FROM  node:7-alpine
WORKDIR  /app
COPY  package.json  /app
RUN  npm  install
```

```
COPY . /app
ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

同样举一反三，Python项目的时候，我们同样可以先拷贝requerements.txt,然后进行pip install requerements.txt，最后再进行COPY 代码。

```
ROM python:3.6
# 创建 app 目录
WORKDIR /app
# 安装 app 依赖
COPY src/requirements.txt ./
RUN pip install -r requirements.txt
# 打包 app 源码
COPY src /app
EXPOSE 8080
CMD [ "python", "server.py" ]
```

12. 设置默认的环境变量，映射端口和数据卷 运行Docker容器时很可能需要一些环境变量。在Dockerfile设置默认的环境变量是一种很好的方式。另外，我们应该在Dockerfile中设置映射端口和数据卷。示例如下: ```dockerfile FROM node:7-alpine ENV PROJECT_DIR=/app WORKDIR PROJECT_DIR RUN npm install COPY . MEDIA_DIR EXPOSE $APP_PORT ENTRYPOINT ["./entrypoint.sh"] CMD ["start"] ``` [ENV] (https://docs.docker.com/engine/reference/builder/#env)指令指定的环境变量在容器中可以使用。如果你只是需要指定构建镜像时的变量，你可以使用[ARG] (https://docs.docker.com/engine/reference/builder/#arg)指令。

13. 使用LABEL设置镜像元数据

使用LABEL指令，可以为镜像设置元数据，例如**镜像创建者**或者**镜像说明**。旧版的Dockerfile语法使用MAINTAINER指令指定镜像创建者，但是它已经被弃用了。有时，一些外部程序需要用到镜像的元数据，例如nvidia-docker需要用到 `com.nvidia.volumes.needed` 。示例如下:

```
FROM node:7-alpine
LABEL maintainer "jakub.skalecki@example.com"
...
```

14. 添加HEALTHCHECK

运行容器时，可以指定 `--restart always` 选项。这样的话，容器崩溃时，Docker守护进程(docker daemon)会重启容器。对于需要长时间运行的容器，这个选项非常有用。但是，如果容器的确在运行，但是不可(陷入死循环，配置错误)用怎么办？使用HEALTHCHECK指令可以让Docker周期性的检查容器的健康状况。我们只需要指定一个命令，如果一切正常的话返回0，否则返回1。对HEALTHCHECK感兴趣的话，可以参考这篇博客。示例如下:

```
FROM node:7-alpine
LABEL maintainer "jakub.skalecki@example.com"
ENV PROJECT_DIR=/app
WORKDIR $PROJECT_DIR
COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR
ENV MEDIA_DIR=/media \
    NODE_ENV=production \
    APP_PORT=3000
VOLUME $MEDIA_DIR
EXPOSE $APP_PORT
HEALTHCHECK CMD curl --fail http://localhost:$APP_PORT || exit 1
ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

当请求失败时， `curl --fail` 命令返回非0状态。

15. 多阶段构建

参考文档《https://docs.docker.com/develop/develop-images/multistage-build/》
在docker不支持多阶段构建的年代，我们构建docker镜像时通常会采用如下两种方法：
方法A.将所有的构建过程编写在同一个Dockerfile中，包括项目及其依赖库的编译、测试、打包等流程，可能会有如下问题：

- **- Dockerfile可能会特别臃肿**
- **- 镜像层次特别深**
- **- 存在源码泄露的风险**

方法B.事先在外部将项目及其依赖库编译测试打包好后，再将其拷贝到构建目录中执行构建镜像。
方法B较方法A略显优雅一些，而且可以很好地规避方法A存在的风险点，但仍需要我们编写两套或多套Dockerfile或者一些脚本才能将其两个阶段自动整合起来，例如有多个项目彼此关联和依赖，就需要我们维护多个Dockerfile，或者需要编写更复杂的脚本，导致后期维护成本很高。

为解决以上问题，**Docker v17.05 开始支持多阶段构建 (multistage builds)**。使用多阶段构建我们就可以很容易解决前面提到的问题，并且只需要编写一个 Dockerfile。
你可以在一个 Dockerfile 中使用多个 FROM 语句。每个 FROM 指令都可以使用不同的基础镜像，并表示开始一个新的构建阶段。你可以很方便的将一个阶段的文件复制到另外一个阶段，在最终的镜像中保留下你需要的内容即可。
默认情况下，构建阶段是没有命令的，我们可以通过它们的索引来引用它们，第一个 FROM 指令从0开始，我们也可以用AS指令为构建阶段命名。

案例1

```
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .FROM alpin
e:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=0 /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

通过 `docker build` 构建后，最终结果是产生与之前相同大小的 Image，但复杂性显著降低。您不需要创建任何中间 Image，也不需要将任何编译结果临时提取到本地系统。
哪它是如何工作的呢？关键就在 `COPY --from=0` 这个指令上。Dockerfile 中第二个 FROM 指令以 alpine:latest 为基础镜像开始了一个新的构建阶段，并通过 `COPY --from=0` 仅将前一阶段的构建文件复制到此阶段。前一构建阶段中产生的 Go SDK 和任何中间层都会在此阶段中被舍弃，而不是保存在最终 Image 中。
使用多阶段构建一个python应用。

案例2

默认情况下，构建阶段是未命名的。您可以通过一个整数值来引用它们，默认是从第 0 个 FROM 指令开始的。 为了方便管理，您也可以通过向 FROM 指令添加 as NAME 来命名您的各个构建阶段。下面的示例就通过命名各个构建阶段并在 COPY 指令中使用名称来访问指定的构建阶段。
这样做的好处就是即使稍后重新排序 Dockerfile 中的指令，COPY 指令一样能找到对应的构建阶段。

```
FROM golang:1.7.3 as builder
WORKDIR /go/src/github.com/alexellis/href-counter/
```

```
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .FROM alpin
e:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

案例3

停在特定的构建阶段

构建镜像时，不一定需要构建整个 Dockerfile 中每个阶段，您也可以指定需要构建的阶段。比如：您只构建 Dockerfile 中名为 builder 的阶段

```
$ docker build --target builder -t alexellis2/href-counter:latest .
```

此功能适合以下场景：

- 调试特定的构建阶段。
- 在 Debug 阶段，启用所有程序调试模式或调试工具，而在生产阶段尽量精简。
- 在 Testing 阶段，您的应用程序使用测试数据，但在生产阶段则使用生产数据。

案例4

使用外部镜像作为构建阶段

使用多阶段构建时，您不仅可以从 Dockerfile 中创建的镜像中进行复制。您还可以使用 `COPY --from` 指令从单独的 Image 中复制，支持使用本地 Image 名称、本地或 Docker 注册中心可用的标记或标记 ID。

```
COPY --from=nginx:latest /etc/nginx/nginx.conf /nginx.conf
```

案例5

把前一个阶段作为一个新的阶段

在使用 FROM 指令时，您可以通过引用前一阶段停止的地方来继续。同样，采用此方式也可以方

便一个团队中的不同角色，如何使用类似流水线的方式，一级一级提供基础镜像，同样更方便快速的复用团队其他人的基础镜像。例如:

```
FROM alpine:latest as builder
RUN apk --no-cache add build-base
FROM builder as build1
COPY source1.cpp source.cpp
RUN g++ -o /binary source.cpp
FROM builder as build2
COPY source2.cpp source.cpp
RUN g++ -o /binary source.cpp
```

```
# ---- 基础 python 镜像 ----
FROM python:3.6 AS base
# 创建 app 目录
WORKDIR /app
# ---- 依赖 ----
FROM base AS dependencies
COPY gunicorn_app/requirements.txt ./
# 安装 app 依赖
RUN pip install -r requirements.txt
# ---- 复制文件并 build ----
FROM dependencies AS build
WORKDIR /app
COPY . /app
# 在需要时进行 Build 或 Compile
# --- 使用 Alpine 发布 ----
FROM python:3.6-alpine3.7 AS release
# 创建 app 目录
WORKDIR /app
COPY --from=dependencies /app/requirements.txt ./
COPY --from=dependencies /root/.cache /root/.cache
# 安装 app 依赖
RUN pip install -r requirements.txt
COPY --from=build /app/ ./
CMD ["gunicorn", "--config", "./gunicorn_app/conf/gunicorn_config.py", "gunicorn_ap
p:app"]
```

公众号：运维开发故事

github：https://github.com/orgs/sunsharing-note/dashboard

博客：https://www.devopstory.cn

爱生活，爱运维

我是冬子先生，《运维开发故事》公众号团队中的一员，一线运维农民工，云原生实践者，这里不仅有硬核的技术干货，还有我们对技术的思考和感悟，欢迎关注我们的公众号，期待和你一起成长！

扫码二维码

关注我，不定期维护优质内容

温馨提示

如果我的文章对你有所帮助，还请帮忙**点赞、在看、转发**一下，你的支持会激励我输出更高质量的文章，非常感谢！

你还可以把我的公众号设为「**星标**」，这样当公众号文章更新时，你会在第一时间收到推送消息，避免错过我的文章更新。

.....