原文：https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

本文档介绍了构建高效镜像的建议最佳实践和方法。

Docker通过读取Dockerfile中的指令自动构建镜像 - Dockerfile是一个文本文件，按顺序包含构建给定镜像所需的所有命令。 Dockerfile遵循特定的格式和指令集，您可以在Dockerfile reference.中找到它们。

Docker镜像由只读层组成，每个层都代表一个Dockerfile指令。这些层是堆叠的，每个层都是前一层变化的增量。考虑这个Dockerfile：

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

每条指令创建一个层：

- `FROM` 从ubuntu：18.04 Docker镜像创建一个层.
- `COPY` 从Docker客户端的当前目录添加文件.
- `RUN` 用make构建你的应用程序。
- `CMD` 指定在容器中运行的命令。

运行镜像并生成容器时，可以在基础层的顶部添加新的可写层（“容器图层”）。对正在运行的容器所做的所有更改（例如，写入新文件，修改现有文件和删除文件）都将写入此可写容器层。

有关镜像图层的更多信息（以及Docker如何构建和存储镜像），请参阅About storage drivers.

一般准则和建议

创建短暂的容器

Dockerfile定义的镜像应该生成尽可能短暂的容器。通过“短暂”，我们的意思是容器可以被停止和销毁，然后重建并用绝对最小的设置和配置替换。

请参阅 *The Twelve-factor App*方法下的 Processes，以了解以无状态方式运行容器的动机。

了解构建上下文

发出docker build命令时，当前工作目录称为构建上下文。默认情况下，假定Dockerfile位于此处，但您可以使用文件标志（-f）指定其他位置。无论Dockerfile实际存在于何处，当前目录中的所有文件和目录的递归内容都将作为构建上下文发送到Docker守护程序。

Build context example

为构建的上下文创建一个目录并cd进入其中。将“hello”写入名为hello的文本文件中，并创建一个在其上运行cat的Dockerfile。从构建上下文（.）中构建镜像：

```
mkdir myproject && cd myproject
echo "hello" > hello
echo -e "FROM busybox\nCOPY /hello /\nRUN cat /hello" > Dockerfile
docker build -t helloapp:v1 .
```

将Dockerfile和hello移动到单独的目录中并构建镜像的第二个版本（不依赖于上一次构建的缓存）。使用-f指向Dockerfile并指定构建上下文的目录：

```
mkdir -p dockerfiles context
mv Dockerfile dockerfiles && mv hello context
docker build --no-cache -t helloapp:v2 -f dockerfiles/Dockerfile context
```

无意中包含构建镜像不必要的文件会导致更大的构建上下文和更大的镜像大小。这会增加构建镜像，拉取和推送镜像的时间以及容器运行时大小。要查看构建上下文有多大，请在构建Dockerfile时查找类似这样的消息：

```
Sending build context to Docker daemon  187.8MB
```

通过stdin管道输送Dockerfile

Docker能够依靠通过stdin管道输送 `Dockerfile` 和本地或远程构建上下文来构建镜像。通过stdin管道输送Dockerfile对于执行一次性构建非常有用，无需将Dockerfile写入磁盘，或者在生成Dockerfile的情况下，并且之后不应该持久化。

为方便起见，本节中的示例使用此处的文档，但可以使用在stdin上提供Dockerfile的任何方法。

例如，以下命令是等效的：

```
echo -e 'FROM busybox\nRUN echo "hello world"' | docker build -
```

```
docker build -<<EOF
FROM busybox
RUN echo "hello world"
EOF
```

您可以使用您的首选方法或最适合您的用例的方法替换示例。

使用来自STDIN的DOCKERFILE构建镜像，且不发送构建上下文

使用此语法用stdin中的Dockerfile构建镜像，而不将其他文件作为构建上下文发送。连字符（ - ）取代PATH的位置，并指示Docker从stdin而不是目录中读取构建上下文（仅包含Dockerfile）：

```
docker build [OPTIONS] -
```

以下示例使用通过stdin传递的Dockerfile构建镜像。没有文件作为构建上下文发送到守护程序。

```
docker build -t myimage:latest -<<EOF
FROM busybox
RUN echo "hello world"
EOF
```

在Dockerfile不需要将文件复制到镜像中时，省略构建上下文非常有用，并且可以提高构建速度，因为没有文件发送到守护程序。

如果要通过从构建上下文中排除某些文件来提高构建速度，请参阅exclude with .dockerignore.

注意：如果使用此语法，则尝试构建使用COPY或ADD的Dockerfile将失败。以下示例说明了这一点：

```
# create a directory to work in
mkdir example
cd example

# create an example file
touch somefile.txt

docker build -t myimage:latest -<<EOF
FROM busybox
COPY somefile.txt .
RUN cat /somefile.txt
EOF

# observe that the build fails
...
Step 2/3 : COPY somefile.txt .
COPY failed: stat /var/lib/docker/tmp/docker-builder249218248/somefile.txt: no such file or directory
```

使用来自STDIN的DOCKERFILE，从本地构建上下文构建

使用此语法在本地文件系统上使用文件构建镜像，但是使用来自stdin的`Dockerfile` ，语法使用-f（或--file）选项指定要使用的Dockerfile，使用连字符（ - ）作为文件名来指示Docker从stdin读取Dockerfile：

```
docker build [OPTIONS] -f- PATH
```

下面的示例使用当前目录(.)作为构建上下文，并使用通过stdin传递的Dockerfile构建镜像，using a here document.

```
# create a directory to work in
mkdir example
cd example

# create an example file
touch somefile.txt

# build and image using the current directory as context, and a Dockerfile passed through stdin
docker build -t myimage:latest -f- . <<EOF
FROM busybox
COPY somefile.txt .
RUN cat /somefile.txt
EOF
```

使用来自STDIN的DOCKERFILE，从远程构建上下文构建

通过此语法使用来自远程git仓库的文件构建镜像，`Dockerfile` 来自于stdin。语法使用-f（或--file）选项指定要使用的Dockerfile，使用连字符（ - ）作为文件名来指示Docker从stdin读取Dockerfile：

```
docker build [OPTIONS] -f- PATH
```

如果您希望从不包含Dockerfile的存储库构建镜像，或者您希望使用自定义Dockerfile构建，而不维护自己的存储库分支，则此语法非常有用。

下面的示例使用stdin中的Dockerfile构建镜像，并添加了 “hello-world” Git repository on GitHub 的 `README.md文件。`

```
docker build -t myimage:latest -f- https://github.com/docker-library/hello-world.git <<EOF
FROM busybox
COPY README.md .
EOF
```

Under the hood

使用远程Git存储库构建镜像作为构建上下文时，Docker会在本地计算机上执行存储库的git克隆，并将这些文件作为构建上下文发送到守护程序。此功能要求在运行docker build命令的主机上安装git。

Exclude with .dockerignore

要排除与构建无关的文件（不重构源存储库），请使用.dockerignore文件。此文件支持类似于.gitignore文件的排除模式。有关创建一个的信息，请参阅.dockerignore file.

使用多阶段构建

Multi-stage builds允许您大幅减小最终镜像的大小，而无需减少中间层和文件的数量。

由于镜像是在构建过程的最后阶段构建的，因此可以通过利用构建缓存来最小化镜像层。

例如，**如果您的构建包含多个层，则可以从较不频繁更改（以确保构建缓存可重用）到更频繁更改的顺序进行排序**：

- 安装构建应用程序所需的工具
- 安装或更新库依赖项
- 生成应用程序

Go应用程序的Dockerfile可能如下所示：

```
FROM golang:1.11-alpine AS build

# Install tools required for project
# Run `docker build --no-cache .` to update dependencies
RUN apk add --no-cache git
RUN go get github.com/golang/dep/cmd/dep

# List project dependencies with Gopkg.toml and Gopkg.lock
# These layers are only re-built when Gopkg files are updated
COPY Gopkg.lock Gopkg.toml /go/src/project/
WORKDIR /go/src/project/
# Install library dependencies
RUN dep ensure -vendor-only

# Copy the entire project and build it
# This layer is rebuilt when a file changes in the project directory
COPY . /go/src/project/
RUN go build -o /bin/project

# This results in a single layer image
FROM scratch
COPY --from=build /bin/project /bin/project
ENTRYPOINT ["/bin/project"]
CMD ["--help"]
```

不要安装不必要的包

为了降低复杂性，依赖性，文件大小和构建时间，请避免安装额外的或不必要的软件包，只是因为它们可能“需要”。例如，您不需要在数据库镜像中包含文本编辑器。

解耦应用程序

每个容器应该只关心一件事情。将应用程序分离到多个容器中可以更容易地水平扩展和重用容器。例如，Web应用程序堆栈可能包含三个独立的容器，每个容器都有自己独特的镜像，以分离的方式管理Web应用程序，数据库和内存缓存。

将每个容器限制为一个进程是一个很好的经验法则，但它不是一个硬性规则。例如，不仅可以使用init进程生成容器，而且某些程序可能会自行生成其他进程。例如，Celery 可以生成多个工作进程，Apache 可以为每个请求创建一个进程。

使用您的最佳判断，尽可能保持容器清洁和模块化。如果容器彼此依赖，则可以使用Docker容器网络来确保这些容器可以进行通信。

最小化层数

在旧版本的Docker中，最大限度地减少镜像中的层数量以确保它们具有高性能非常重要。添加了以下功能以减少此限制：

- 只有RUN，COPY，ADD指令才能创建图层。其他指令创建临时中间镜像，并不增加构建的大小。
- 在可能的情况下，使用多阶段构建，并仅将所需的东西复制到最终镜像中。这允许您在中间构建阶段中包含工具和调试信息，而不会增加最终镜像的大小。

对多行参数进行排序

只要有可能，通过按字母顺序排序多行参数来缓解以后的更改。这有助于避免重复包并使列表更容易更新。这也使PR更容易阅读和审查。在反斜杠（\）之前添加空格也有帮助。

这是 `buildpack-deps` image 中的一个示例：

```
RUN apt-get update && apt-get install -y \
  bzr \
  cvs \
```

```
git \
mercurial \
subversion
```

利用构建缓存

构建镜像时，Docker会逐步执行Dockerfile中的指令，按指定的顺序执行每个指令。在检查每条指令时，Docker会在其缓存中查找可以重用的现有镜像，而不是创建新的（重复）镜像。

如果您根本不想使用缓存，可以在docker build命令中使用--no-cache = true选项。但是，如果您确实让Docker使用其缓存，那么了解它何时可以找到匹配的镜像非常重要。 Docker遵循的基本规则概述如下：

- 从已经在高速缓存中的父镜像开始，将下一条指令会将该基本镜像导出的所有子镜像进行比较，以查看它们中的一个是否使用完全相同的指令构建。如果不是，则缓存无效。
- 在大多数情况下，只需将Dockerfile中的指令与其中一个子镜像进行比较即可。但是，某些说明需要更多的检查和解释。
- 对于ADD和COPY指令，将检查镜像中文件的内容，并为每个文件计算校验和。在这些校验和中不考虑文件的最后修改时间和最后访问时间。在缓存查找期间，将校验和与现有镜像中的校验和进行比较。如果文件中的任何内容（例如内容和元数据）发生了任何更改，则缓存将失效。
- 除了ADD和COPY命令之外，缓存检查不会查看容器中的文件以确定高速缓存匹配。例如，在处理RUN apt-get -y update命令时，不检查容器中更新的文件以确定是否存在缓存命中。在这种情况下，只需使用命令字符串本身来查找匹配项。

缓存无效后，所有后续Dockerfile命令都会生成新镜像像，并且不使用缓存。

Dockerfile指令

这些建议旨在帮助您创建高效且可维护的Dockerfile。

FROM

Dockerfile reference for the FROM instruction

尽可能使用当前的官方镜像作为镜像的基础。我们推荐Alpine image ，**因为它是严格控制的并且尺寸小（目前小于5 MB），同时仍然是完整的Linux发行版。**

LABEL

Understanding object labels

您可以为镜像添加标签，以帮助按项目组织镜像，记录许可信息，帮助实现自动化或出于其他原因。对于每个标签，添加以LABEL开头并带有一个或多个键值对的行。以下示例显示了不同的可接受格式。内容包括解释性意见。

必须引用带空格的字符串或必须转义空格。内引号字符（“）也必须转义。

```
# Set one or more individual labels
LABEL com.example.version="0.0.1-beta"
LABEL vendor1="ACME Incorporated"
LABEL vendor2=ZENITH\ Incorporated
LABEL com.example.release-date="2015-02-12"
LABEL com.example.version.is-production=""
```

镜像可以有多个标签。在Docker 1.10之前，建议将所有标签组合到单个LABEL指令中，以防止创建额外的层。这不再是必需的，但仍然支持组合标签。

```
# Set multiple labels on one line
LABEL com.example.version="0.0.1-beta" com.example.release-date="2015-02-12"
```

以上也可以写成：

```
# Set multiple labels at once, using line-continuation characters to break long lines
LABEL vendor=ACME\ Incorporated \
```

```
    com.example.is-beta= \
    com.example.is-production="" \
    com.example.version="0.0.1-beta" \
    com.example.release-date="2015-02-12"
```

有关可接受的标签键和值的指导，请参阅了解对象标签。有关查询标签的信息，请参阅管理对象标签中与过滤相关的项目。另请参见Dockerfile参考中的LABEL。

RUN

Dockerfile reference for the RUN instruction

在使用反斜杠拆分长或复杂的RUN语句成多行，以使Dockerfile更具可读性，可理解性和可维护性。

APT-GET

RUN最常见的用例可能是apt-get的应用。因为它用来安装软件包，所以RUN apt-get命令有几个需要注意的问题。

避免 RUN apt-get upgrade 和 dist-upgrade，因为父镜像中的许多“基本”包无法在非特权容器内升级。如果父镜像中包含的包已过期，请与其维护人员联系。如果你知道有一个需要更新的特定包foo，请使用apt-get install -y foo自动更新。

始终将RUN apt-get update与apt-get install结合在同一个RUN语句中。例如：

```
RUN apt-get update && apt-get install -y \
    package-bar \
    package-baz \
    package-foo
```

在RUN语句中单独使用apt-get update会导致缓存问题和后续的apt-get install指令失败。例如，假设你有一个Dockerfile：

```
FROM ubuntu:18.04
RUN apt-get update
RUN apt-get install -y curl
```

构建映像后，所有层都在Docker缓存中。假设您稍后通过添加额外的包来修改apt-get install：

```
FROM ubuntu:18.04
RUN apt-get update
RUN apt-get install -y curl nginx
```

Docker将初始和修改的指令视为相同，并重用前面步骤中的缓存。因此，不会执行apt-get更新，因为构建使用缓存版本。由于apt-get update未运行，因此您的构建可能会获得curl和nginx软件包的过时版本。

使用RUN apt-get update && apt-get install -y可确保您的Dockerfile安装最新的软件包版本，无需进一步编码或手动干预。这种技术被称为“缓存破坏”。您还可以通过指定包版本来实现缓存清除。这称为版本固定，例如：

```
RUN apt-get update && apt-get install -y \
    package-bar \
    package-baz \
    package-foo=1.3.*
```

版本固定强制构建以检索特定版本，而不管缓存中的内容是什么。此技术还可以减少由于所需包中意外更改而导致的故障。

下面是一个结构良好的RUN指令，它演示了所有apt-get建议。

```
RUN apt-get update && apt-get install -y \
    aufs-tools \
    automake \
    build-essential \
    curl \
    dpkg-sig \
    libcap-dev \
    libsqlite3-dev \
    mercurial \
    reprepro \
```

```
    ruby1.9.1 \
    ruby1.9.1-dev \
    s3cmd=1.1.* \
&& rm -rf /var/lib/apt/lists/*
```

s3cmd参数指定版本1.1。*。如果镜像以前使用的是旧版本，则指定新版本会导致apt-get update缓存失效，并确保安装新版本。在每行列出包也可以防止包名重复的错误。

此外，当您通过删除/ var / lib / apt / lists清理apt缓存时，它会减小映像大小，因为apt缓存不存储在层中。由于RUN语句以apt-get update开头，因此在apt-get install之前始终刷新包缓存。

注意：官方Debian和Ubuntu映像自动运行apt-get clean，因此不需要显式调用。

使用管道

某些RUN命令依赖于使用管道符（|）将一个命令的输出传递到另一个命令的能力，如下例所示：

```
RUN wget -O - https://some.site | wc -l > /number
```

Docker使用/ bin / sh -c解释器执行这些命令，该解释器仅评估管道中最后一个操作的退出代码以确定成功。在上面的示例中，只要wc -l命令成功，即使wget命令失败，此构建步骤也会成功并生成新映像.

如果希望命令由于管道中任何阶段的错误而失败，前置 set -o pipefail &&，以确保意外错误可防止构建无意中成功。例如：

```
RUN set -o pipefail && wget -O - https://some.site | wc -l > /number
```

并非所有shell都支持-o pipefail选项。

在基于Debian的镜像像上的 `dash` shell的情况下，请考虑使用RUN的exec形式来明确选择支持pipefail选项的shell。例如

```
RUN ["/bin/bash", "-c", "set -o pipefail && wget -O - https://some.site | wc -l > /number"]
```

CMD

```
Dockerfile reference for the CMD instruction
```

CMD指令应该用于运行镜像包含的软件以及所有参数。 CMD应该几乎总是以CMD [“executable”， “param1”， “param2”.....]的形式使用。

因此，如果镜像用于服务，例如Apache和Rails，则可以运行类似CMD [“apache2”， “ - DFOREGROUND”]的内容。

实际上，建议将这种形式的指令用于任何基于服务的镜像。

在大多数其他情况下，CMD应该被赋予一个交互式shell，例如bash，python和perl。例如，CMD [“perl”，“ - de0”]，CMD [“python”]或CMD [“php”，“ - a”]。使用这个格式意味着当你执行像docker run -it python这样的东西时，你将被放入一个准备好了的可用的shell中。 CMD应该很少以CMD [“param”，“param”]的方式与ENTRYPOINT一起使用，除非您和您的预期用户已经非常熟悉ENTRYPOINT的工作原理。

EXPOSE

Dockerfile reference for the EXPOSE instruction

EXPOSE指令指明容器侦听用于连接的端口。因此，您应该为您的应用程序使用通用的传统端口。例如，包含Apache Web服务器的映像将使用EXPOSE 80，而包含MongoDB的映像将使用EXPOSE 27017等。

对于外部访问，您的用户可以使用标志执行docker run，该标志指示如何将指定端口映射到他们选择的端口。对于容器链接，Docker为从接收容器返回源容器的路径提供环境变量（例如MYSQL_PORT_3306_TCP）。

ENV

Dockerfile reference for the ENV instruction

为了使新软件更易于运行，您可以使用ENV更新容器安装的软件的PATH环境变量。例如，ENV PATH /usr/local/nginx/bin:$PATH确保CMD [“nginx”]正常工作。

ENV指令对于提供特定于您希望容纳的服务的必需环境变量也很有用，例如Postgres的PGDATA。

最后，ENV还可用于设置常用版本号，以便更容易维护版本变更，如以下示例所示：

```
ENV PG_MAJOR 9.3
ENV PG_VERSION 9.3.4
RUN curl -SL http://example.com/postgres-$PG_VERSION.tar.xz | tar -xJC /usr/src/postgress && …
ENV PATH /usr/local/postgres-$PG_MAJOR/bin:$PATH
```

与在程序中使用常量变量（与硬编码值相反）类似，此方法允许您更改单个ENV指令以自动神奇地变更容器中的软件版本。

每条ENV线都会创建一个新的中间层，就像RUN命令一样。这意味着即使在将来的层中取消设置环境变量，它仍然会在此层中保留，并且可以转储其值。可以通过创建如下所示的Dockerfile来测试它，然后构建它。

```
FROM alpine
ENV ADMIN_USER="mark"
RUN echo $ADMIN_USER > ./mark
RUN unset ADMIN_USER
```

```
$ docker run --rm test sh -c 'echo $ADMIN_USER'

mark
```

要防止这种情况，并且真的取消设置环境变量，请使用带有shell命令的RUN命令，在单个层中设置、使用和取消设置变量。您可以使用; or &&分割命令。如果您使用第二种方法，并且其中一个命令失败，则docker构建也会失败。这通常是一个好主意。使用\作为Linux Dockerfiles的行继续符可以提高可读性。您还可以将所有命令放入shell脚本中，并使用RUN命令运行该shell脚本。

```
FROM alpine
RUN export ADMIN_USER="mark" \
    && echo $ADMIN_USER > ./mark \
    && unset ADMIN_USER
CMD sh
```

```
$ docker run --rm test sh -c 'echo $ADMIN_USER'
```

ADD or COPY

- Dockerfile reference for the ADD instruction
- Dockerfile reference for the COPY instruction

尽管ADD和COPY在功能上相似，但一般来说，COPY是优选的。那是因为它比ADD更明确。 COPY仅支持基本复制功能，将本地文件复制到容器中，而ADD具有一些其它功能（如仅限本地的tar提取和远程URL支持），这些功能并不是很明确。因此，**ADD的最佳用途是将本地tar文件自动提取到镜像中**，如`ADD rootfs.tar.xz /` 。

如果Dockerfile有多个步骤，使用上下文中不同文件的，请单独复制它们，而不是一次复制它们。这可确保每个步骤的构建缓存仅在特定所需文件更改时失效（强制重新执行该步骤）。

例如：

```
COPY requirements.txt /tmp/
RUN pip install --requirement /tmp/requirements.txt
COPY . /tmp/
```

如果把`COPY . /tmp/`放在RUN 之前，RUN的缓存失效几率更小

由于镜像大小很重要，因此强烈建议不要使用ADD从远程URL获取包。你应该使用curl或wget代替。这样，您可以删除提取后不再需要的文件，也不必在镜像中添加其他图层。例如，你应该避免做以下事情：

```
ADD http://example.com/big.tar.xz /usr/src/things/
RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things
RUN make -C /usr/src/things all
```

推荐这样做：

```
RUN mkdir -p /usr/src/things \
    && curl -SL http://example.com/big.tar.xz \
    | tar -xJC /usr/src/things \
    && make -C /usr/src/things all
```

对于不需要ADD命令自动解包的文件、目录，应该始终使用COPY。

ENTRYPOINT

Dockerfile reference for the ENTRYPOINT instruction

ENTRYPOINT的最佳用途是设置镜像的主命令，允许该镜像像该命令需要的一样运行（然后使用CMD作为默认标志）。

让我们从命令行工具s3cmd的镜像示例开始：

```
ENTRYPOINT ["s3cmd"]
CMD ["--help"]
```

现在可以像这样运行**镜像**来显示命令的帮助：

```
$ docker run s3cmd
```

或使用正确的参数执行命令：

```
$ docker run s3cmd ls s3://mybucket
```

这很有用，因为**镜像**名称可以兼作二进制文件的引用，如上面的命令所示。

ENTRYPOINT指令也可以与辅助脚本结合使用，使其能够以与上述命令类似的方式运行，即使启动该工具可能需要多个步骤。

例如，Postgres官方镜像使用以下脚本作为其ENTRYPOINT：

```
#!/bin/bash
set -e

if [ "$1" = 'postgres' ]; then
    chown -R postgres "$PGDATA"

    if [ -z "$(ls -A "$PGDATA")" ]; then
        gosu postgres initdb
    fi

    exec gosu postgres "$@"
fi

exec "$@"
```

Configure app as PID 1

此脚本使用exec Bash命令，以便最终运行的应用程序成为容器的PID 1.这允许应用程序接收发送到容器的任何Unix信号。有关更多信息，请参阅ENTRYPOINT参考。

帮助脚本被复制到容器中，并且通过`ENTRYPOINT`在氢气启动时运行：

```
COPY ./docker-entrypoint.sh /
ENTRYPOINT ["/docker-entrypoint.sh"]
CMD ["postgres"]
```

该脚本允许用户以多种方式与Postgres交互。

它可以简单地启动Postgres：

```
$ docker run postgres
```

或者，它可用于运行Postgres并将参数传递给服务器：

```
$ docker run postgres postgres --help
```

最后，它还可以用来启动一个完全不同的工具，比如Bash：

```
$ docker run --rm -it postgres bash
```

VOLUME

Dockerfile reference for the VOLUME instruction

VOLUME指令应用于公开由docker容器创建的所有数据库存储区域，配置存储或文件/文件夹。强烈建议您将VOLUME用于镜像的任何可变和/或用户可维修部分。

USER

Dockerfile reference for the USER instruction

如果服务可以在没有权限的情况下运行，请使用USER更改为非root用户。首先在Dockerfile中创建用户和组，例如 `RUN groupadd -r postgres && useradd --no-log-init -r -g postgres postgres`.

Consider an explicit UID/GID

镜像中的用户和组被分配了非确定性UID / GID，因此无论镜像如何重建，都会分配“下一个”UID / GID。因此，如果它很重要，您应该分配一个显式的UID / GID。

由于Go archive/ tar 包处理稀疏文件时未解决的bug，尝试在Docker容器内创建具有非常大的UID的用户可能导致磁盘耗尽，因为容器层中的/var/log/faillog填充满了NULL（\ 0）字符。解决方法是将--no-log-init标志传递给useradd。 Debian / Ubuntu adduser wrapper 不支持此标志。

避免安装或使用sudo，因为它具有不可预测的TTY和可能导致问题的信号转发行为。如果您绝对需要与sudo类似的功能，例如将守护程序初始化为root但将其作为非root运行，请考虑使用“gosu”。
最后，为了减少层数和复杂性，请避免频繁地来回切换USER。

WORKDIR

Dockerfile reference for the WORKDIR instruction

为了清晰和可靠，您应该始终使用WORKDIR的绝对路径。此外，您应该使用WORKDIR而不是使用 `RUN cd … && do-something` 这样的指令，这些指令难以阅读，故障排除和维护。

ONBUILD

Dockerfile reference for the ONBUILD instruction

在当前Dockerfile构建完成后执行ONBUILD命令。 ONBUILD在 FROM 当前镜像 派生的任何子镜像中执行。将ONBUILD命令视为父Dockerfile为子Dockerfile提供的指令。

Docker构建在子Dockerfile中的任何命令之前执行ONBUILD命令。

ONBUILD对于将从给定镜像构建的镜像非常有用。例如，您可以使用ONBUILD作为语言堆栈镜像，在Dockerfile中构建使用该语言编写的任意用户软件，如Ruby的ONBUILD变体中所示。

从ONBUILD构建的镜像应该获得一个单独的标记，例如：ruby：1.9-onbuild或ruby：2.0-onbuild。

将ADD或COPY放入ONBUILD时要小心。如果新构建的上下文缺少正在添加的资源，则“onbuild”映像将发生灾难性故障。如上所述，添加单独的标记有助于通过允许Dockerfile作者做出选择来减少这种情况。

官方镜像的例子

These Official Images have exemplary `Dockerfile`s:

- Go
- Perl
- Hy

- Ruby

其他资源：

- Dockerfile Reference
- More about Base Images
- More about Automated Builds
- Guidelines for Creating Official Images