注意，在dockerfile中每条指令都是独立运行的，并且会导致创建新镜像 - 因此RUN cd / tmp对下一条指令不会产生任何影响。

FROM

```
FROM <image> [AS <name>]
```

Or

```
FROM <image>[:<tag>] [AS <name>]
```

Or

```
FROM <image>[@<digest>] [AS <name>]
```

该FROM指令初始化新的构建阶段并为后续指令设置 基本镜像。因此，有效Dockerfile必须以FROM指令开头。镜像可以是任何有效镜像 - 通过从公共存储库中提取镜像来启动

- ARG 在Dockerfile是唯一可能处在From之前的指令。请参阅了解ARG和FROM如何交互。
- FROM 可以在单个Dockerfile镜像中多次出现以创建多个镜像，或者使用一个构建阶段作为另一个构建阶段的依赖项。只需在每条新FROM指令之前记下提交输出的最后一个
- 通过在From指令后面添加AS name 可以给一个新的构建命名，该操作可选。该名称可用于后续FROM和 COPY --from=<name|index>指示，以引用此阶段构建的镜像。
- tag或digest值是可选的。如果省略其中任何一个，则构建器默认采用latest标记。如果找不到tag值，构建器将返回错误。

了解ARG和FROM如何互动

FROM指令支持所有在第一个From之前由ARG指令声明的变量。

```
ARG  CODE_VERSION=latest
FROM base:${CODE_VERSION}
CMD  /code/run-app

FROM extras:${CODE_VERSION}
CMD  /code/run-extras
```

在FROM之前声明的ARG指令处于构建阶段之外，因此它不能在FROM之后的任何指令使用。要使用在第一次FROM之前声明的ARG变量，需要在构建阶段中使用一个不带参数的ARG指令：

```
ARG VERSION=latest
FROM busybox:$VERSION
ARG VERSION
RUN echo $VERSION > image_version
```

RUN

run有两种格式：

- RUN <command> (*shell* form, the command is run in a shell, which by default is /bin/sh -c on Linux or cmd /S /C on Windows)
- RUN ["executable", "param1", "param2"] (*exec* form)

RUN指令将在当前镜像之上的新层中执行任何命令并提交结果。生成的已提交镜像将用于Dockerfile中的下一步。

分层RUN指令和生成提交符合Docker的核心概念，其中提交很容易，并且可以从镜像历史中的任何点创建容器，就像源代码控制一样。

exec格式可以防止shell的字符串改写，并且使用基础镜像RUN命令，而不包含指定的shell可执行文件

可以使用SHELL命令更改shell格式的默认shell。

在shell形式中，您可以使用\（反斜杠）将单个RUN指令继续到下一行。例如：

```
RUN /bin/bash -c 'source $HOME/.bashrc; \
echo $HOME'
```

它们一起相当于这一行：

```
RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME'
```

注意：要使用与'/bin/sh'不同的shell解释器，在*exec* 格式中需要传入指定的shell，例如：

```
RUN ["/bin/bash", "-c", "echo hello"]
```

注意：exec表单被解析为JSON数组，这意味着您必须使用双引号（“）来围绕单词而不是单引号（'）。

注意：与shell表单不同，exec表单不会调用命令shell。这意味着不会发生正常的shell处理。例如，RUN [“echo”，“$ HOME”]不会对$ HOME执行变量替换。如果你想要shell处理，那么要么使用shell表单，要么直接执行shell，例如：RUN [“sh”，“ - c”，“echo $ HOME”]。当使用exec表单并直接执行shell时（如前面的例子），它是shell执行环境变量扩展，而不是docker。

注意：在JSON表单中，必须转义反斜杠。这在反斜杠是路径分隔符的Windows上尤为重要。由于不是有效的JSON，以下行将被视为shell表单，并以意外方式失败：RUN ["c:\windows\system32\tasklist.exe"]，此示例的正确语法是：RUN ["c:\\windows\\system32\\tasklist.exe"]

在下一次构建期间，RUN指令的缓存不会自动失效。像RUN apt-get dist-upgrade -y这样的指令的缓存将在下一次构建期间重用。可以使用--no-cache标志使RUN指令的高速缓存无效，例如docker build --no-cache。

ADD指令可以使RUN指令的高速缓存无效。请参阅 下文了解详情。

CMD

该CMD指令有三种形式：

- `CMD ["executable","param1","param2"]` (*exec* form, this is the preferred form)
- `CMD ["param1","param2"]` (as *default parameters to ENTRYPOINT*)
- `CMD command param1 param2` (*shell* form)

Dockerfile中只能有一条CMD指令。如果列出多个CMD，则只有最后一个CMD才会生效。

CMD的主要目的是为执行容器提供默认值。这些默认值可以包含可执行文件，也可以省略可执行文件，在这种情况下，您还必须指定ENTRYPOINT指令。

注意：如果使用CMD为ENTRYPOINT指令提供默认参数，则应使用JSON数组格式指定CMD和ENTRYPOINT指令。

注意：exec表单被解析为JSON数组，这意味着您必须使用双引号（“）来围绕单词而不是单引号（'）。

在shell或exec格式中使用时，CMD指令设置运行镜像时要执行的命令。如果你使用CMD的shell形式，那么<command>将在/ bin / sh -c中执行：

```
FROM ubuntu
CMD echo "This is a test." | wc -
```

如果要在没有shell的情况下运行<command>，则必须将该命令表示为JSON数组并提供可执行文件的完整路径。此数组形式是CMD的首选格式。任何其他参数必须在数组中单独表

```
FROM ubuntu
CMD ["/usr/bin/wc","--help"]
```

如果您希望容器每次都运行相同的可执行文件，那么您应该考虑将ENTRYPOINT与CMD结合使用。请参阅ENTRYPOINT.

如果用户指定了docker run的参数，那么它们将覆盖CMD中指定的默认值。

注意：不要将RUN与CMD混淆。 RUN实际上运行一个命令并提交结果；CMD在构建时不执行任何操作，但指定了镜像的预期命令。

LABEL

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

LABEL指令将元数据添加到镜像。 LABEL是键值对。要在LABEL值中包含空格，请使用引号和反斜杠，就像在命令行解析中一样。一些用法示例：

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

镜像可以有多个标签。您可以在一行中指定多个标签。在Docker 1.10之前，这减小了最终镜像的大小，但现在不再是这种情况了。您仍然可以选择在单个指令中指定多个标签，方法有以下两种：

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

```
LABEL multi.label1="value1" \
      multi.label2="value2" \
      other="value3"
```

基础或父镜像中包含的标签（FROM行中的镜像）由镜像继承。如果标签已存在但具有不同的值，则最近应用的值将覆盖任何先前设置的值。

要查看镜像的标签，请使用docker inspect命令。

```
"Labels": {
    "com.example.vendor": "ACME Incorporated"
    "com.example.label-with-value": "foo",
    "version": "1.0",
    "description": "This text illustrates that label-values can span multiple lines.",
    "multi.label1": "value1",
    "multi.label2": "value2",
    "other": "value3"
},
```

MAINTAINER (已弃用)

```
MAINTAINER <name>
```

MAINTAINER指令设置生成镜像的Author字段。 LABEL指令是一个更加灵活的版本，您应该使用它，因为它可以设置您需要的任何元数据，并且可以轻松查看，例如使用

docker inspect。要设置与MAINTAINER字段对应的标签，您可以使用：

```
LABEL maintainer="SvenDowideit@home.org.au"
```

然后，这将从docker inspect与其他标签一起显示。

EXPOSE

```
EXPOSE <port> [<port>/<protocol>...]
```

EXPOSE指令通知Docker容器在运行时侦听指定的网络端口。您可以指定端口是侦听TCP还是UDP，如果未指定协议，则默认为TCP。

EXPOSE指令实际上不会发布端口。它在构建镜像的人和运行容器的人之间起到一种文档的作用，说明关于哪些端口要发布。要在运行容器时实际发布端口，请在docker run上使用-p标志发布和映射一个或多个端口，或使用-P标志发布所有公开的端口并将它们映射到高阶端口。

默认情况下，EXPOSE假定为TCP。您还可以指定UDP：

```
EXPOSE 80/udp
```

要同时暴露在TCP和UDP协议，需要两行：

```
EXPOSE 80/tcp
EXPOSE 80/udp
```

在这种情况下，如果将-P与docker run一起使用，则端口将为TCP公开一次，对UDP公开一次。请记住，-P在主机上使用短暂的高阶主机端口，因此TCP和UDP的端口不同。

无论EXPOSE设置如何，您都可以使用-p标志在运行时覆盖它们。例如

```
docker run -p 80:80/tcp -p 80:80/udp ...
```

要在主机系统上设置端口重定向，请参阅使用-P标志。该`docker network`命令支持为容器之间的通信创建网络，而无需公开或发布特定端口，因为连接到网络的容器可以通过任何端口相互通信。有关详细信息，请参阅此功能的 概述 ）。

ENV

```
ENV <key> <value>
ENV <key>=<value> ...
```

该`ENV`指令将环境变量`<key>`设置为该值 `<value>`。此值将在构建阶段中的所有后续指令的环境中，并且也可以在许多内联替换。

ENV指令有两种形式。第一种形式ENV <key> <value>，将单个变量设置为一个值。第一个空格后面的整个字符串将被视为<value> - 包括空格字符。该值将针对其他环境变量进行解释，因此如果未对其进行转义，则将删除引号字符。

第二种形式ENV <key> = <value> ...允许一次设置多个变量。请注意，第二种形式在语法中使用等号（ = ），而第一种形式则不然。与命令行解析一样，引号和反斜杠可用于在值内包含空格。

例如：

```
ENV myName="John Doe" myDog=Rex\ The\ Dog \
    myCat=fluffy
```

and

```
ENV myName John Doe
ENV myDog Rex The Dog
ENV myCat fluffy
```

将在最终镜像中产生相同的净结果。

当从生成的镜像运行容器时，使用ENV设置的环境变量将保持不变。您可以使用docker inspect查看值，并使用docker run --env <key> = <value>更改它们。

注意：环境持久性可能会导致意外的副作用。例如，设置ENV DEBIAN_FRONTEND noninteractive可能会使基于Debian的镜像上的apt-get用户感到困惑。要为单个命令设置值，请使用RUN <key> = <value> <command>。

ADD

ADD有两种形式：

- `ADD [--chown=<user>:<group>] <src>... <dest>`
- `ADD [--chown=<user>:<group>] ["<src>",... "<dest>"]` (路径包含空格时，必须使用这种格式)

注意：--chown功能仅在用于构建Linux容器的Dockerfiles上受支持，并且不适用于Windows容器。由于用户和组所有权概念不能在Linux和Windows之间进行转换，因此使用/etc/passwd和/etc/group将用户名和组名转换为ID会限制此功能仅适用于基于Linux OS的容器。

ADD指令从<src>复制新文件，目录或远程文件URL，并将它们添加到镜像文件系统中的<dest>路径。

可以指定多个<src>资源，但如果它们是文件或目录，则它们的路径将被解释为构建上下文的源的相对路径。（ Multiple `<src>` resources may be specified but if they are files or directories, their paths are interpreted as relative to the source of the context of the build. ）

每个<src>可能包含通配符，匹配将使用Go的filepath.Match规则完成。例如：

```
ADD hom* /mydir/        # adds all files starting with "hom"
ADD hom?.txt /mydir/    # ? is replaced with any single character, e.g., "home.txt"
```

<dest>是绝对路径，或相对于WORKDIR的路径，源将复制到目标容器的该路径中。

添加包含特殊字符（例如[和]）的文件或目录时，需要按照Golang规则转义这些路径，以防止它们被视为匹配模式。例如，要添加名为arr[0].txt的文件，请使用以下命令：

```
ADD arr[[]0].txt /mydir/    # copy a file named "arr[0].txt" to /mydir/
```

除非可选的--chown标志指定了特定的用户名，组名或UID / GID组合以请求添加内容的特定所有权，否则将使用UID和GID为0创建所有新文件和目录。

--chown标志的格式允许用户名和组名字符串或直接整数UID和GID的任意组合，提供没有组名的用户名或没有GID的UID，GID将使用与UID相同的数字.如果提供了用户名或组名，则容器的根文件系统/etc/passwd和/etc/group文件将分别用于执行从名称到整数UID或GID的转换。以下示例显示了--chown标志的有效定义：

```
ADD --chown=55:mygroup files* /somedir/
ADD --chown=bin files* /somedir/
ADD --chown=1 files* /somedir/
ADD --chown=10:11 files* /somedir/
```

如果容器根文件系统不包含/ etc / passwd或/ etc / group文件，并且在--chown标志中使用了用户名或组名，则构建将在ADD操作上失败。使用数字ID不需要查找，也不依赖于容器根文件系统内容。

在<src>是远程文件URL的情况下，目标将具有600的权限。如果正在检索的远程文件具有HTTP Last-Modified 的header，则该header的时间戳将用于设置目标上的mtime文件。但是，与ADD期间处理的任何其他文件一样，mtime将不包含在判断文件是否已更改和缓存应该更新。

注意：如果通过STDIN传递Dockerfile（`docker build - < somefile`）来构建，则没有构建上下文，因此Dockerfile只能包含基于URL的ADD指令。您还可以通过STDIN传递压缩文件:(`docker build - < archive.tar.gz`），Dockerfile在压缩文件的根目录下，文件的其余部分将用作构建的上下文。

注意：如果您的URL文件使用身份验证进行保护，则需要使用`RUN wget`, `RUN curl`或使用容器内的其他工具，因为ADD指令不支持身份验证。

注意：如果`<src>`内容已更改，则第一个遇到的ADD指令将使Dockerfile的所有后续指令的高速缓存无效。这包括使RUN指令的高速缓存无效。有关详细信息，请参阅“`Dockerfile`最佳实践指南”。【建议：**因此尽量将ADD指令放在Dockerfile的后面**】

***ADD*遵守以下规则：**

- **<src>路径必须位于构建的上下文中**;你不能 `ADD ../something /something`，因为docker构建的第一步是将上下文目录（和子目录）发送到docker守护进程。
- 如果<src>是URL且<dest>不以尾部斜杠结尾，则从URL下载文件并将其复制到<dest>。【推测是将文件下载后重命名为<.dest>文件，未验证】
- 如果<src>是URL并且<dest>以尾部斜杠结尾，则从URL推断文件名，并将文件下载到<dest>/<filename>。例如，`ADD http://example.com/foobar /` 将创建文件/foobar。URL必须具有非常重要的路径，以便在这种情况下可以发现适当的文件名（http://example.com将不起作用）。【将URL文件下载到<dest>/目录】
- 如果<src>是目录，则复制目录的全部内容，包括文件系统元数据。注意：不复制目录本身，只复制其内容。
- 如果<src>是可识别的压缩格式（identity，gzip，bzip2或xz）的本地tar文件，则将其解压缩为目录。远程URL中的资源不会被解压缩。复制或解压缩目录时，它与tar -x具有相同的行为，结果是以下的集合：

  a. 无论在目标地路径上存在什么，

  b. 源目录树的内容，包括在逐个文件的基础上，得益于“2.”被解决的冲突

  注意：文件是否被标识为可识别的压缩格式仅基于文件的内容而不是文件的名称来完成。例如，如果空文件恰好以.tar.gz结尾，则不会将其识别为压缩文件，也不会生成任何类型的解压缩错误消息，而是将文件简单地复制到目标。
- 如果<src>是任何其他类型的文件，则将其与元数据一起单独复制。在这种情况下，如果<dest>以尾部斜杠/结尾，则将其视为目录，<src>的内容将写入<dest>/base(<src>)。
- 如果直接或由于使用通配符指定了多个<src>资源，则<dest>必须是目录，并且必须以斜杠/结尾。
- 如果<dest>不以尾部斜杠结束，则将其视为常规文件，<src>的内容将写入<dest>。
- 如果<dest>不存在，则会在其路径中创建所有缺少的目录。

为什么有了*ADD*，还要*COPY*？

目前ADD指令有点让人迷惑，有时候解压文件，有时候不解压文件，如果你想拷贝一个压缩文件，你会以为地解压。如果文件是某种不能识别的压缩文件，如果你想解压，你又会意外地复制它。

在Docker 1.0发布时候，包括了新指令COPY。不像是ADD，COPY 更加直接了当，只复制文件或者目录到容器里。

COPY不支持URL，也不会特别对待压缩文件。如果build 上下文件中没有指定解压的话，那么就不会自动解压，只会复制压缩文件到容器中。

COPY是ADD的一种简化版本，目的在于满足大多数人“复制文件到容器”的需求。

Docker 团队的建议是在大多数情况下使用COPY

COPY

COPY有两种形式：

- `COPY [--chown=<user>:<group>] <src>... <dest>`
- `COPY [--chown=<user>:<group>] ["<src>",... "<dest>"]` (路径包含空格时，必须使用这种格式)

注意：--chown功能仅在用于构建Linux容器的Dockerfiles上受支持，并且不适用于Windows容器。由于用户和组所有权概念不能在Linux和Windows之间进行转换，因此使用/etc/passwd和/etc/group将用户名和组名转换为ID会限制此功能仅适用于基于Linux OS的容器。

COPY指令从<src>复制新文件或目录，并将它们添加到容器的文件系统路径<dest>路径。

可以指定多个<src>资源，但文件和目录的路径将被解释为构建上下文的源的相对路径。（Multiple `<src>` resources may be specified but the paths of files and directories will be interpreted as relative to the source of the context of the build.）

每个都<src>可能包含通配符，匹配将使用Go的 filepath.Match规则完成。例如：

```
COPY hom* /mydir/        # adds all files starting with "hom"
COPY hom?.txt /mydir/    # ? is replaced with any single character, e.g., "home.txt"
```

<dest>是绝对路径，或相对于WORKDIR的路径，源将在目标容器中复制到该路径中。

```
COPY test relativeDir/   # adds "test" to `WORKDIR`/relativeDir/
COPY test /absoluteDir/  # adds "test" to /absoluteDir/
```

复制包含特殊字符（例如 [和] ）的文件或目录时，需要按照Golang规则转义这些路径，以防止它们被视为匹配模式。例如，要复制名为arr [0] .txt的文件，请使用以下命令：

```
COPY arr[[]0].txt /mydir/    # copy a file named "arr[0].txt" to /mydir/
```

除非可选的--chown标志指定了特定的用户名，组名或UID / GID组合以请求添加内容的特定所有权，否则将使用UID和GID为0创建所有新文件和目录。

--chown标志的格式允许用户名和组名字符串或直接整数UID和GID的任意组合，提供没有组名的用户名或没有GID的UID，GID将使用与UID相同的数字.如果提供了用户名或组名，则容器的根文件系统/etc/passwd和/etc/group文件将分别用于执行从名称到整数UID或GID的转换。以下示例显示了--chown标志的有效定义：

```
COPY --chown=55:mygroup files* /somedir/
COPY --chown=bin files* /somedir/
COPY --chown=1 files* /somedir/
COPY --chown=10:11 files* /somedir/
```

如果容器根文件系统不包含/ etc / passwd或/ etc / group文件，并且在--chown标志中使用了用户名或组名，则构建将在COPY操作上失败。使用数字ID不需要查找，也不依赖于容器根文件系统内容。

注意：如果使用STDIN（docker build - < somefile）构建，则没有构建上下文，因此无法使用COPY。

可选地，COPY接受一个标志 --from=<name|index>，可用于将源位置设置为先前的构建阶段（使用 FROM .. AS <name>创建），而不是由用户发送的构建上下文用户。该标志还接受由FROM指令开头的所有先前构建阶段分配的数字索引。如果找不到具有指定名称的构建阶段，则尝试使用具有相同名称的镜像。

***COPY*遵守以下规则：**

- <src>路径必须位于构建的上下文中;你不能 COPY ../something /something，因为docker构建的第一步是将上下文目录（和子目录）发送到docker守护进程。
- 如果<src>是目录，则复制目录的全部内容，包括文件系统元数据，注意：不复制目录本身，只复制其内容。
- 如果<src>是任何其他类型的文件，则将其与元数据一起单独复制。在这种情况下，如果<dest>以尾部斜杠/结尾，则将其视为目录，<src>的内容将写入<dest>/base(<src>)。
- 如果直接或由于使用通配符指定了多个<src>资源，则<dest>必须是目录，并且必须以斜杠/结尾。
- 如果<dest>不以尾部斜杠结束，则将其视为常规文件，<src>的内容将写入<dest>。
- 如果<dest>不存在，则会在其路径中创建所有缺少的目录。

ENTRYPOINT

ENTRYPOINT有两种形式：

- `ENTRYPOINT ["executable", "param1", "param2"]` (*exec* form, preferred)
- `ENTRYPOINT command param1 param2` (*shell* form)

ENTRYPOINT允许您配置将作为可执行文件运行的容器。

例如，以下将使用其默认内容启动nginx，侦听端口80：

```
docker run -i -t --rm -p 80:80 nginx
```

docker run <image> 的命令行参数将会被添加到 ENTRYPOINT的exec格式所有元素之后，并且将会覆盖由CMD指定的所有元素。这允许将参数传递给entrypoint，例如docker run <image> -d 将会把 -d 参数传递给entrypoint，您可以使用 docker run --entrypoint标志覆盖ENTRYPOINT指令。

shell格式可防止使用任何CMD或run命令行参数，但缺点是 ENTRYPOINT将作为不会传递信号的 /bin/sh -c的子命令启动，这意味着可执行文件不是容器的PID 1 -- 并且不会收到Unix信号 -- 因此您的可执行文件不会从docker stop <container>接收SIGTERM。

只有Dockerfile中的最后一个ENTRYPOINT指令才会生效。

Exec form ENTRYPOINT example

可以使用ENTRYPOINT的exec形式设置相当稳定的默认命令和参数，然后使用任一形式的CMD来设置更可能更改的其他默认值。

```
FROM ubuntu
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
```

运行容器时，您可以看到top是唯一的进程：

```
$ docker run -it --rm --name test  top -H
```

```
top - 08:25:00 up  7:27,  0 users,  load average: 0.00, 0.01, 0.05
Threads:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   2056668 total,  1616832 used,   439836 free,    99352 buffers
KiB Swap:  1441840 total,        0 used,  1441840 free.  1324440 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
    1 root      20   0   19744   2336   2080 R  0.0  0.1   0:00.04 top
```

要进一步检查结果，可以使用docker exec：

```
$ docker exec -it test ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  2.6  0.1  19752  2352 ?        Ss+  08:24   0:00 top -b -H
root         7  0.0  0.1  15572  2164 ?        R+   08:25   0:00 ps aux
```

并且您可以优雅地使用 docker stop test来请求关闭 top。

以下Dockerfile显示使用ENTRYPOINT在前台运行Apache（即，作为PID 1）：

```
FROM debian:stable
RUN apt-get update && apt-get install -y --force-yes apache2
EXPOSE 80 443
VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

如果需要为单个可执行文件编写启动脚本，可以使用 exec 和 gosu 命令确保最终的可执行文件接收Unix信号：

```
#!/usr/bin/env 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 "$@"
```

最后，如果您需要在关机时进行一些额外的清理（或与其他容器通信），或者协调多个可执行文件，您可能需要确保ENTRYPOINT脚本接收Unix信号，传递它们，然后做一些更多的工作：

```
#!/bin/sh
# Note: I've written this using sh so it works in the busybox container too

# USE the trap if you need to also do manual cleanup after the service is stopped,
#     or need to start multiple services in the one container
trap "echo TRAPed signal" HUP INT QUIT TERM

# start service in background here
/usr/sbin/apachectl start

echo "[hit enter key to exit] or run 'docker stop <container>'"
read

# stop service and clean up here
echo "stopping apache"
/usr/sbin/apachectl stop

echo "exited $0"
```

如果你使用docker run -it --rm -p 80:80 --name test apache运行这个镜像，你可以用docker exec或docker top检查容器的进程，然后让脚本停止Apache：

```
$ docker exec -it test ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.0   4448   692 ?        Ss+  00:42   0:00 /bin/sh /run.sh 123 cmd cmd2
root        19  0.0  0.2  71304  4440 ?        Ss   00:42   0:00 /usr/sbin/apache2 -k start
www-data    20  0.2  0.2 360468  6004 ?        Sl   00:42   0:00 /usr/sbin/apache2 -k start
www-data    21  0.2  0.2 360468  6000 ?        Sl   00:42   0:00 /usr/sbin/apache2 -k start
root        81  0.0  0.1  15572  2140 ?        R+   00:44   0:00 ps aux
$ docker top test
PID                 USER                COMMAND
10035               root                {run.sh} /bin/sh /run.sh 123 cmd cmd2
```

```
10054               root               /usr/sbin/apache2 -k start
10055               33                 /usr/sbin/apache2 -k start
10056               33                 /usr/sbin/apache2 -k start
$ /usr/bin/time docker stop test
test
real    0m 0.27s
user    0m 0.03s
sys     0m 0.03s
```

注意：您可以使用--entrypoint覆盖ENTRYPOINT设置，但这只能将二进制设置为exec（不使用sh -c）。

注意：exec表单被解析为JSON数组，这意味着您必须使用双引号（“）来围绕单词而不是单引号（'）。

注意：与shell格式不同，exec格式不会调用命令shell。这意味着不会发生正常的shell处理。例如，ENTRYPOINT [“echo”，“$ HOME”]不会对$ HOME执行变量替换。如果你想要shell处理，那么要么使用shell表单，要么直接执行shell，例如：ENTRYPOINT [“sh”，“ - c”，“echo $ HOME”]。当使用exec表单并直接执行shell时（如shell表单的情况），它是执行环境变量扩展的shell，而不是docker。

Shell form ENTRYPOINT example

您可以为ENTRYPOINT指定一个纯字符串，它将在 /bin/sh -c 中执行。此格式将使用shell处理来替换shell环境变量，并将忽略任何CMD或 docker run命令行参数。要确保docker stop能正确发出信号到长时间运行的具有ENTRYPOINT的容器中，你需要记住用exec启动它：

```
FROM ubuntu
ENTRYPOINT exec top -b
```

运行此镜像时，您将看到单个PID 1进程：

```
$ docker run -it --rm --name test top
Mem: 1704520K used, 352148K free, 0K shrd, 0K buff, 140368121167873K cached
CPU:   5% usr   0% sys   0% nic  94% idle   0% io   0% irq   0% sirq
Load average: 0.08 0.03 0.05 2/98 6
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
    1     0 root     R     3164   0%   0% top -b
```

并且将会在 docker stop 时干净的退出

```
$ /usr/bin/time docker stop test
test
real    0m 0.20s
user    0m 0.02s
sys     0m 0.04s
```

如果您忘记将exec添加到ENTRYPOINT指令的开头：

```
FROM ubuntu
ENTRYPOINT top -b
CMD --ignored-param1
```

然后，您可以运行它（为下一步命名）：

```
$ docker run -it --name test top --ignored-param2
Mem: 1704184K used, 352484K free, 0K shrd, 0K buff, 140621524238337K cached
CPU:   9% usr   2% sys   0% nic  88% idle   0% io   0% irq   0% sirq
Load average: 0.01 0.02 0.05 2/101 7
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
    1     0 root     S     3168   0%   0% /bin/sh -c top -b cmd cmd2
    7     1 root     R     3164   0%   0% top -b
```

您可以从top的输出中看到指定的ENTRYPOINT不是PID 1。

如果然后运行docker stop test，容器将不会干净地退出 -- `stop` 命令将被强制在超时后发送SIGKILL：

```
$ docker exec -it test ps aux
PID   USER     COMMAND
    1 root     /bin/sh -c top -b cmd cmd2
    7 root     top -b
    8 root     ps aux
$ /usr/bin/time docker stop test
test
real    0m 10.19s
user    0m 0.04s
sys     0m 0.03s
```

了解CMD和ENTRYPOINT如何相互作用

CMD和ENTRYPOINT指令都定义了在运行容器时执行的命令。以下几条规则描述他们的合作：

1. Dockerfile应至少指定一个CMD或ENTRYPOINT命令。
2. 使用容器作为可执行文件时，应定义ENTRYPOINT。

3. CMD应该用作为ENTRYPOINT命令定义默认参数或在容器中执行ad-hoc命令的方法
4. 使用备用参数运行容器时，将覆盖CMD

下表显示了针对不同ENTRYPOINT / CMD组合执行的命令：

	No ENTRYPOINT	ENTRYPOINT exec_entry p1_entry	ENTRYPOINT [“exec_entry”, “p1_entry”]
No CMD	*error, not allowed*	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD [“exec_cmd”, “p1_cmd”]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD [“p1_cmd”, “p2_cmd”]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

注意：如果从基本镜像定义CMD，则设置ENTRYPOINT会将CMD重置为空值。在这种情况下，必须在当前镜像中定义CMD才能获得值。

VOLUME

```
VOLUME ["/data"]
```

VOLUME指令创建具有指定名称的挂载点，并将其标记为从本机主机或其他容器保存外部安装的卷。该值可以是JSON数组，`VOLUME ["/var/log/"]`，或具有多个参数的普通字符串，例如`VOLUME /var/log`或`VOLUME /var/log /var/db`。有关通过Docker客户端提供的更多信息/示例和安装说明，请参阅 通过卷共享目录 文档。

docker run命令使用基础镜像中指定位置存在的任何数据初始化新创建的卷。例如，参考以下Dockerfile片段：

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
```

此Dockerfile会生成一个镜像，该镜像会导致docker run在 /myvol上创建新的挂载点，并将greeting文件复制到新创建的卷中。

有关指定卷的说明

关于Dockerfile中的卷，请记住以下事项。

1. **基于Windows的容器上的卷**：使用基于Windows的容器时，容器中卷的目标必须是以下之一：
   - 不存在或空目录
   - C：以外的驱动器
1. **从Dockerfile中更改卷**：如果任何构建步骤在声明后更改卷内的数据，那么这些更改将被丢弃。
2. **JSON格式**：列表被解析为JSON数组。您必须用双引号（“）而不是单引号（'）括起来
3. **主机目录在容器运行时声明**：主机目录（mountpoint）本质上是依赖于主机的。这是为了保持镜像的可移植性，因为不能保证给定的主机目录在所有主机上都可用。因此，您无法从Dockerfile中安装主机目录。 VOLUME指令不支持指定host-dir参数。**您必须在创建或运行容器时指定挂载点。**

USER

```
USER <user>[:<group>] or
USER <UID>[:<GID>]
```

USER指令设置用户名（或UID）以及可选的用户组（或GID），以便在运行**镜像**时以及Dockerfile中跟随它的任何RUN，CMD和ENTRYPOINT指令时使用。

警告：当用户没有primary group时，将使用root组运行镜像（或下一条指令）

在Windows上，如果用户不是内置帐户，则必须先创建用户。这可以使用作为Dockerfile一部分 调用的net user命令来完成

```
FROM microsoft/windowsservercore
# Create Windows user in the container
RUN net user /add patrick
# Set it for subsequent commands
USER patrick
```

WORKDIR

```
WORKDIR /path/to/workdir
```

WORKDIR指令为Dockerfile中的任何RUN，CMD，ENTRYPOINT，COPY和ADD指令设置工作目录。如果WORKDIR不存在，即使它未在任何后续Dockerfile指令中使用，也将创建它。

WORKDIR指令可以在Dockerfile中多次使用。如果提供了相对路径，则它将相对于先前WORKDIR指令的路径。例如：

```
WORKDIR /a
WORKDIR b
WORKDIR c
```

```
RUN pwd
```

此Dockerfile中最终pwd命令的输出为 `/a/b/c`。

WORKDIR指令可以解析先前使用ENV设置的环境变量。您只能使用Dockerfile中显式设置的环境变量。例如：

```
ENV DIRPATH /path
WORKDIR $DIRPATH/$DIRNAME
RUN pwd
```

此Dockerfile中最后一个pwd命令的输出将是 `/path/$DIRNAME`

ARG

```
ARG <name>[=<default value>]
```

ARG指令使用 `--build-arg <varname>=<value>` 标志定义一个变量，用户可以使用docker build命令在构建时将该变量传递给构建器。如果用户指定了未在Dockerfile中定义的构建参数，则构建会输出警告。

```
[Warning] One or more build-args [foo] were not consumed.
```

Dockerfile可以包括一个或多个ARG指令。例如，以下是有效的Dockerfile：

```
FROM busybox
ARG user1
ARG buildno
...
```

警告：建议不要使用构建时变量来传递github密钥，用户凭据等秘密。使用docker history命令可以使镜像的任何用户都可以看到构建时变量值。

Default values

ARG指令可以选择包含默认值：

```
FROM busybox
ARG user1=someuser
ARG buildno=1
...
```

如果ARG指令具有默认值，并且在构建时没有传递值，则构建器将使用默认值。

Scope

ARG变量定义从Dockerfile中它被定义的行开始生效，而不是从使用它的命令行或其他地方。例如，考虑这个Dockerfile：

```
1 FROM busybox
2 USER ${user:-some_user}
3 ARG user
4 USER $user
...
```

用户通过调用以下内容构建此文件：

```
$ docker build --build-arg user=what_user .
```

第2行的USER赋值为some_user，因为user变量在后续第3行才定义。第4行的USER赋值为what_user，因为 user已经被定义，并且通过命令行把`what_user` 值传递过来。在通过ARG指令定义之前，对变量的任何使用都会是空字符串。

ARG指令在它被定义的那个构建阶段结束时超出范围。要在多个阶段中使用arg，每个阶段必须包含ARG指令。eg:

```
FROM busybox
ARG SETTINGS
RUN ./run/setup $SETTINGS

FROM busybox
ARG SETTINGS
RUN ./run/other $SETTINGS
```

Using ARG variables

您可以使用ARG或ENV指令指定RUN指令可用的变量。使用ENV指令定义的环境变量始终覆盖同名的ARG指令。参考这个带有ENV和ARG指令的Dockerfile。

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 ENV CONT_IMG_VER v1.0.0
4 RUN echo $CONT_IMG_VER
```

然后，假设使用此命令构建此镜像：

```
$ docker build --build-arg CONT_IMG_VER=v2.0.1 .
```

在这种情况下，RUN指令使用v1.0.0而不是用户传递的ARG设置：v2.0.1；此行为类似于shell脚本，其中本地范围的变量覆盖作为参数传递或从环境继承的变量，从它的定义来看。

使用上面的示例，但不同的ENV格式，您可以在ARG和ENV指令之间创建更有用的交互：

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 ENV CONT_IMG_VER ${CONT_IMG_VER:-v1.0.0}
4 RUN echo $CONT_IMG_VER
```

与ARG指令不同，ENV值始终保留在构建的镜像中。考虑没有--build-arg标志的docker构建：

```
$ docker build .
```

使用此Dockerfile示例，CONT_IMG_VER仍然保留在镜像中，但其值为v1.0.0，因为它是ENV指令在第3行中的默认设置。

此示例中的变量扩展技术允许您从命令行传递参数，并通过利用该ENV指令将它们保存在最终镜像中 。只有一组有限的Dockerfile指令支持变量扩展。

Predefined ARGs

Docker有一组预定义的ARG变量，您可以在Dockerfile中使用而无需相应的ARG指令。

- `HTTP_PROXY`
- `http_proxy`
- `HTTPS_PROXY`
- `https_proxy`
- `FTP_PROXY`
- `ftp_proxy`
- `NO_PROXY`
- `no_proxy`

要使用它们，只需使用标志在命令行上传递它们：

```
--build-arg <varname>=<value>
```

默认情况下，这些预定义变量将从docker history的输出中排除。排除它们可以降低在HTTP_PROXY变量中意外泄露敏感验证信息的风险。

例如，考虑构建以下Dockerfile使用 --build-arg HTTP_PROXY=http://user:pass@proxy.lon.example.com

```
FROM ubuntu
RUN echo "Hello World"
```

在这种情况下，HTTP_PROXY变量的值在docker历史记录中不可用，并且不会被缓存。如果要更改位置，并且代理服务器已更改为http://user:pass@proxy.sfo.example.com，则后续构建不会导致缓存未命中。

如果您需要覆盖此行为，则可以通过在Dockerfile中添加ARG语句来执行此操作，如下所示：

```
FROM ubuntu
ARG HTTP_PROXY
RUN echo "Hello World"
```

构建此Dockerfile时，HTTP_PROXY将保留在docker历史记录中，并且更改其值会使构建缓存无效。

Automatic platform ARGs in the global scope

此功能仅在使用BuildKit后端时可用。

Docker预定义了一组ARG变量，其中包含执行构建（构建平台）的节点平台上的信息以及生成的镜像（目标平台）的平台上的信息。可以使用docker build上的--platform标志指定目标平台。

以下ARG变量自动设置：

- `TARGETPLATFORM` - platform of the build result. Eg `linux/amd64`, `linux/arm/v7`, `windows/amd64`.
- `TARGETOS` - OS component of TARGETPLATFORM
- `TARGETARCH` - architecture component of TARGETPLATFORM
- `TARGETVARIANT` - variant component of TARGETPLATFORM
- `BUILDPLATFORM` - platform of the node performing the build.
- `BUILDOS` - OS component of BUILDPLATFORM
- `BUILDARCH` - OS component of BUILDPLATFORM
- `BUILDVARIANT` - OS component of BUILDPLATFORM

这些参数在全局范围内定义，因此在构建阶段或RUN命令中不会自动提供。要在构建阶段中使用其中一个参数，需要不带值的重新定义。例如：

```
FROM alpine
ARG TARGETPLATFORM
RUN echo "I'm building for $TARGETPLATFORM"
```

对构建缓存的影响

ARG变量不会像ENV变量那样持久保存到构建的镜像中。但是，ARG变量会以类似的方式影响构建缓存。如果Dockerfile定义了一个ARG变量，它的值与先前构建不同，则在第一次使用时会发生“缓存未命中”，而不是在定义时。特别是，ARG指令之后的所有RUN指令都隐式地使用ARG变量（作为环境变量），因此可能导致高速缓存未命中。除非Dockerfile中存在匹配的ARG声明，否则所有预定义的ARG变量都将不被缓存。

例如，这两个Dockerfile：

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 RUN echo $CONT_IMG_VER
```

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 RUN echo hello
```

如果在命令行上指定 `--build-arg CONT_IMG_VER=<value>`，则在这两种情况下，第2行上的规范都不会导致高速缓存未命中;第3行确定导致缓存未命中.ARG CONT_IMG_VER导致RUN行被识别为与运行 `CONT_IMG_VER=<value>` echo hello相同，因此如果<value>发生更改，将获得缓存未命中。

同一命令行下的另一个示例：

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 ENV CONT_IMG_VER $CONT_IMG_VER
4 RUN echo $CONT_IMG_VER
```

在此示例中，高速缓存未命中发生在第3行。由于ENV中的变量值引用ARG变量并且该变量通过命令行更改，因此发生未命中。在此示例中，ENV命令使镜像包含该值。

如果ENV指令覆盖了同名的ARG指令，如此Dockerfile：

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 ENV CONT_IMG_VER hello
4 RUN echo $CONT_IMG_VER
```

第3行不会导致缓存未命中，因为CONT_IMG_VER的值是常量（hello）。因此，RUN（第4行）上使用的环境变量和值在构建之间不会发生变化。

ONBUILD

```
ONBUILD [INSTRUCTION]
```

当**镜像**用作另一个构建的基础时，ONBUILD指令向**镜像**添加将在稍后执行的触发指令。触发器将在下游构建的上下文中执行，就好像它是在下游Dockerfile中的FROM指令之后立即插入的一样。

任何构建指令都可以注册为触发器。

如果要构建将用作构建其他**镜像**的基础的**镜像**，这非常有用，例如应用程序构建环境或用户自定义配置的deamon。

例如，如果您的**镜像**是可重用的Python应用程序构建器，则需要将应用程序源代码添加到特定目录中，并且可能需要在此之后调用构建脚本。您现在不能只调用ADD和RUN，因为您还无法访问应用程序源代码，并且每个应用程序构建都会有所不同。您可以简单地为应用程序开发人员提供一个样板Dockerfile来复制粘贴到他们的应用程序中，但这样做效率低，容易出错且难以更新，因为它与特定于应用程序的代码混合在一起。

解决方案是使用ONBUILD来注册预先指令，以便在下一个构建阶段运行。

以下是它的工作原理：

1. 当遇到ONBUILD指令时，构建器会为正在构建的图像的元数据添加触发器。该指令不会影响当前构建。
2. 在构建结束时，所有触发器的列表都存储在映像清单中的OnBuild键下。可以使用docker inspect命令检查它们。
3. 稍后，可以使用FROM指令将图像用作新构建的基础。作为处理FROM指令的一部分，下游构建器查找ONBUILD触发器，并按照它们注册的顺序执行它们。如果任何触发器失败，则中止FROM指令，这反过来导致构建失败。如果所有触发器都成功，则FROM指令完成，并且构建继续照常进行。
4. 执行后，触发器将从最终图像中清除。换句话说，它们不是由“grand-children”构建继承的。

例如，您可以添加以下内容：

```
[...]
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
[...]
```

注意：ONBUILD ONBUILD这样的链式指令是不被允许的

警告：ONBUILD指令可能不会触发FROM或MAINTAINER指令。

STOPSIGNAL

```
STOPSIGNAL signal
```

STOPSIGNAL指令设置系统调用将发送到容器的信号以退出。此信号可以是与内核的系统调用表中的位置匹配的有效无符号数，例如9，或SIGNAME格式的信号名，例如SIGKILL。

HEALTHCHECK

HEALTHCHECK指令有两种形式：

- `HEALTHCHECK [OPTIONS] CMD command` (通过在容器内运行命令来检查容器运行状况)
- `HEALTHCHECK NONE` (禁用从基础镜像继承的任何健康检查)

HEALTHCHECK指令告诉Docker如何测试容器以检查它是否仍在工作。即使服务器进程仍在运行，这也可以检测到陷入无限循环且无法处理新连接的Web服务器等情况。

当容器指定了 healthcheck 时，除了正常状态外，它还具有*health status*。此状态初始状态为starting。每当健康检查通过时，它就会变成`healthy` （无论以前处于什么状态）。经过一定数量的连续失败后，它变成unhealthy。

可以在CMD命令之前出现的选项是：

- `--interval=DURATION` (default: `30s`)
- `--timeout=DURATION` (default: `30s`)
- `--start-period=DURATION` (default: `0s`)
- `--retries=N` (default: `3`)

运行状况检查将首先在容器启动后的 **interval** 秒运行，然后在每次上一次检查完成后再 **interval** 隔秒。

如果检查的单次运行时间超过超时秒数，则认为检查失败。

它需要重试连续的健康检查失败才能认为容器 unhealthy。

start period为需要时间引导的容器提供初始化时间。在此期间探测失败将不计入最大重试次数。但是，如果在启动期间运行状况检查成功，则会将容器视为已启动，并且所有连续失败将计入最大重试次数。

Dockerfile中只能有一个HEALTHCHECK指令。如果列出多个，那么只有最后一个HEALTHCHECK才会生效。

CMD关键字之后的命令可以是shell命令（例如HEALTHCHECK CMD / bin / check-running）或exec数组（与其他Dockerfile命令一样;有关详细信息，请参阅例如ENTRYPOINT）。

命令的退出状态指示容器的运行状况。可能的值是：

- 0: success - 容器是健康的，随时可以使用
- 1: unhealthy - 容器无法正常工作
- 2: reserved（保留） - 不要使用此退出代码

例如，要每五分钟检查网络服务器能够在三秒钟内为网站的主页面提供服务：

```
HEALTHCHECK --interval=5m --timeout=3s \
  CMD curl -f http://localhost/ || exit 1
```

为了帮助调试失败的探测器，命令在stdout或stderr上写入的任何输出文本（UTF-8编码）都将存储在运行状况中，并可以使用docker inspect进行查询。此类输出应保持较短（目前仅存储前4096个字节）。

当容器的运行状况更改时，将生成具有新状态的health_status事件。

在Docker 1.12中添加了HEALTHCHECK功能。

SHELL

```
SHELL ["executable", "parameters"]
```

SHELL指令允许覆盖用于shell形式命令的默认shell。 Linux上的默认shell是["/bin/sh", "-c"]，在Windows上是["cmd", "/S", "/C"]。 SHELL指令必须以JSON格式写入Dockerfile。

SHELL指令在Windows上特别有用，因为windows有两个常用且完全不同的本机shell：cmd和powershell，以及包括sh的备用shell。

SHELL指令可以多次出现。每个SHELL指令都会覆盖所有先前的SHELL指令，并影响所有后续指令。例如：

```
FROM microsoft/windowsservercore

# Executed as cmd /S /C echo default
RUN echo default

# Executed as cmd /S /C powershell -command Write-Host default
RUN powershell -command Write-Host default

# Executed as powershell -command Write-Host hello
SHELL ["powershell", "-command"]
RUN Write-Host hello

# Executed as cmd /S /C echo hello
SHELL ["cmd", "/S", "/C"]
RUN echo hello
```

当在Dockerfile中使用 RUN，CMD和ENTRYPOINT 的shell形式时SHELL指令可能会影响它们。

以下示例是在Windows上找到的常见模式，可以使用SHELL指令简化：

```
...
RUN powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"
...
```

docker调用的命令将是：

```
cmd /S /C powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"
```

由于两个原因，这是低效的。首先，调用一个不必要的cmd.exe命令处理器（也就是shell）。其次，shell形式的每个RUN指令都需要一个额外的powershell -command前缀命令。

为了提高效率，可以采用两种机制中的一种。一种是使用RUN命令的JSON形式，例如：

```
...
RUN ["powershell", "-command", "Execute-MyCmdlet", "-param1 \"c:\\foo.txt\""]
...
```

虽然JSON表单是明确的，并且不使用不必要的cmd.exe，但它确实需要通过双引号和转义更加详细。备用机制是使用SHELL指令和shell格式，为Windows用户提供更自然的语法，特别是与 `escape` 解释器直接结合使用时：

```
# escape=`

FROM microsoft/nanoserver
SHELL ["powershell","-command"]
RUN New-Item -ItemType Directory C:\Example
ADD Execute-MyCmdlet.ps1 c:\example\
RUN c:\example\Execute-MyCmdlet -sample 'hello world'
```

结果：

```
PS E:\docker\build\shell> docker build -t shell .
Sending build context to Docker daemon 4.096 kB
Step 1/5 : FROM microsoft/nanoserver
 ---> 22738ff49c6d
Step 2/5 : SHELL powershell -command
 ---> Running in 6fcdb6855ae2
 ---> 6331462d4300
Removing intermediate container 6fcdb6855ae2
Step 3/5 : RUN New-Item -ItemType Directory C:\Example
 ---> Running in d0eef8386e97


    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        10/28/2016  11:26 AM                Example


 ---> 3f2fbf1395d9
Removing intermediate container d0eef8386e97
Step 4/5 : ADD Execute-MyCmdlet.ps1 c:\example\
 ---> a955b2621c31
Removing intermediate container b825593d39fc
Step 5/5 : RUN c:\example\Execute-MyCmdlet 'hello world'
 ---> Running in be6d8e63fe75
hello world
 ---> 8e559e9bf424
Removing intermediate container be6d8e63fe75
Successfully built 8e559e9bf424
PS E:\docker\build\shell>
```

SHELL指令也可用于修改shell的运行方式。例如，在Windows上使用 SHELL cmd /S /C /V:ON|OFF，可以修改延迟的环境变量扩展语义。

如果需要备用shell，例如zsh，csh，tcsh等，也可以在Linux上使用SHELL指令。
在Docker 1.12中添加了SHELL功能。

外部实现功能

此功能仅在使用BuildKit后端时可用。

Docker构建支持实验性功能，如缓存挂载，构建机密和ssh转发，这些功能是通过使用带有语法指令的构建器的外部实现来启用的。要了解这些功能，请参阅BuildKit存储库中的文档。

Dockerfile examples

下面你可以看到一些Dockerfile语法的例子。如果您对更现实的东西感兴趣，请查看Docker化示例列表。

```
# Nginx
#
# VERSION               0.0.1

FROM      ubuntu
LABEL Description="This image is used to start the foobar executable" Vendor="ACME Products" Version="1.0"
RUN apt-get update && apt-get install -y inotify-tools nginx apache2 openssh-server

# Firefox over VNC
#
# VERSION               0.3

FROM ubuntu

# Install vnc, xvfb in order to create a 'fake' display and firefox
RUN apt-get update && apt-get install -y x11vnc xvfb firefox
RUN mkdir ~/.vnc
# Setup a password
RUN x11vnc -storepasswd 1234 ~/.vnc/passwd
# Autostart firefox (might not be the best way, but it does the trick)
RUN bash -c 'echo "firefox" >> /.bashrc'

EXPOSE 5900
CMD    ["x11vnc", "-forever", "-usepw", "-create"]

# Multiple images example
#
# VERSION               0.1

FROM ubuntu
RUN echo foo > bar
# Will output something like ===> 907ad6c2736f

FROM ubuntu
RUN echo moo > oink
# Will output something like ===> 695d7793cbe4

# You'll now have two images, 907ad6c2736f with /bar, and 695d7793cbe4 with
# /oink.
```

来源： https://docs.docker.com/engine/reference/builder/