- Home

Gotchas in Writing Dockerfile

Contents of This Article

Why do we need to use Dockerfile?

Dockerfile is not yet-another shell. Dockerfile has its special mission: **automation of Docker image creation.**

Once, you write build instructions into Dockerfile, you can build the same image just with `docker build` command.

Dockerfile is also useful to tell the knowledge of what a job the container does to somebody else. Your teammates can tell what the container is supposed to do just by reading Dockerfile. They don't need to know login to the container and figure out what the container is doing by using ps command.

For these reasons, you **must** use Dockerfile when you build images. However, writing Dockerfile is sometimes painful. In this post, I will write a few tips and gochas in writing Dockerfile so that you love the tool.

ADD and understanding context in Dockerfile

ADD is the instruction to add local files to Docker image. The basic usage is very simple. Suppose you want to add a local file called *myfile.txt* to */myfile.txt* of image.

```
$ ls
Dockerfile  mydir  myfile.txt
```

Then your Dockerfile looks like this.

```
ADD myfile.txt /
```

Very simple. However, if you want to add */home/vagrant/myfile.txt*, you can’t do this.

```
# Your have this in your Dockerfile
# ADD /home/vagrant/myfile.txt /

$ docker build -t blog .
Uploading context 10240 bytes
Step 1 : FROM ubuntu
 ---> 8dbd9e392a96
Step 2 : ADD /home/vagrant/myfile.txt /
Error build: /home/vagrant/myfile.txt: no such file or directory
```

You got `no such file or directory` error even if you have the file. Why? This is because */home/vagrant/myfile.txt* is not added to the **context** of Dockerfile. Context in Dockerfile means files and directories available to the Dockerfile instructions. Only files and directories in the context can be added during build. Files and sub directories under the current directory are added to the context. You can see this when you run build command.

```
$ docker build -t blog .
Uploading context 10240 bytes
```

What’s happening here is Docker client makes tarball of entries under the current directory and send it to Docker daemon. The reason why thiis is required is because your Docker daemon may be running on remote machine. That’s why the above command says *Uploading*.

There is a pitfall, though. Since automatically entries under current directories are added to the context, it tries to upload huge files and take longer time for build even if you don’t add the file.

```
$ ls
Dockerfile  very_huge_file

$ docker build -t blog .
Uploading context xxxxxx bytes
..... # Takes very long time
```

So the best practice is only placing files and directories that you need to add to image under current directory.

Treat your container like a binary with CMD

By using CMD instruction in Dockerfile, your container acts like a single executable binary. Suppose you have these instructions in your Dockerfile.

```
# Suppose you have run.sh in the same directory as the Dockerfile
ADD run.sh /usr/local/bin/run.sh
CMD ["/usr/local/bin/run.sh"]
```

When you build a container from this Dockerfile and run with `docker run -i run_image`, it runs `/usr/local/bin/run.sh` script and exists.

If you don't use `CMD`, you always have to pass the command to the argument: `docker run -i run_image /usr/local/bin/run.sh`.

This is not just cumbersome, but also considered to be a bad practice from the perspective of operation.

If you have `CMD` instruction, the purpose of the container becomes explicit: all what the container wants to do is running the command.

But, if you don't have the instruction, anybody except the person who made the container need to rely on external documentation to know how to run the container properly.

So, in general, you should have `CMD` instruction in your Dockerfile.

Difference between CMD and ENTRYPOINT

`CMD` and `ENTRYPOINT` are confusing.

Every commands, either passed as an argument or specified from `CND` instruction are passed as argument of binary specified in `ENTRYPOINT`.

`/bin/sh -c` is the default entrypoint. So if you specify `CMD date` without specifying entrypoint, Docker executes it as `/bin/sh -c date`.

By using entrypoint, you can change the behaviour of your container at run time that makes container operation a bit more flexible.

```
ENTRYPOINT ["/bin/date"]
```

With the entrypoint above, the container prints out current date with different format.

```
$ docker run -i clock_container +"%s"
1404214000

$ docker run -i clock_container +"%F"
2014-07-01
```

exec format error

There is one caveat in default entrypoint. For example, you want to execute the following shell script.

/usr/local/bin/run.sh

```
echo "hello, world"
```

Dockerfile

```
ADD run.sh /usr/local/bin/run.sh
RUN chmod +x /usr/local/bin/run.sh
CMD ["/usr/local/bin/run.sh"]
```

When you run the container, your expectation is the container prints out `hello, world`. However, what you will get is a error message that doesn't make sense.

```
$ docker run -i hello_world_image
2014/07/01 10:53:57 exec format error
```

You see this message when you didn't put shebang in your script, and because of that, default entrypoint `/bin/sh -c` does not know how to run the script.

To fix this, you can either add shebang

/usr/local/bin/run.sh

```
#!/bin/bash
echo "hello, world"
```

or you can specify from command line.

```
$ docker run -entrypoint="/bin/bash" -i hello_world_image
```

Build caching: what invalids cache and not?

Docker creates a commit for each line of instruction in Dockerfile. As long as you don't change the instruction, Docker thinks it doesn't need to change the image, so use cached image which is used by the next instruction as a parent image. This is the reason why `docker build` takes long time in the first time, but immediately finishes in the second time.

```
$ time docker build -t blog .
Uploading context 10.24 kB
Step 1 : FROM ubuntu
 ---> 8dbd9e392a96
Step 2 : RUN apt-get update
 ---> Running in 15705b182387
Ign http://archive.ubuntu.com precise InRelease
```

```
Hit http://archive.ubuntu.com precise Release.gpg
Hit http://archive.ubuntu.com precise Release
Hit http://archive.ubuntu.com precise/main amd64 Packages
Get:1 http://archive.ubuntu.com precise/main i386 Packages [1641 kB]
Get:2 http://archive.ubuntu.com precise/main TranslationIndex [3706 B]
Get:3 http://archive.ubuntu.com precise/main Translation-en [893 kB]
Fetched 2537 kB in 7s (351 kB/s)

 ---> a8e9f7328cc4
Successfully built a8e9f7328cc4

real	0m8.589s
user	0m0.008s
sys	0m0.012s

$ time docker build -t blog .
Uploading context 10.24 kB
Step 1 : FROM ubuntu
 ---> 8dbd9e392a96
Step 2 : RUN apt-get update
 ---> Using cache
 ---> a8e9f7328cc4
Successfully built a8e9f7328cc4

real	0m0.067s
user	0m0.012s
sys	0m0.000s
```

However, when cache is used and what invalids cache are sometimes not very clear. Here is a few cases that I found worth to note.

Cache invalidation at one instruction invalids cache of all subsequent instructions

This is the basic rule of caching. If you cause cache invalidation at one instruction, subsequent instructions doesn't use cache.

```
# Before
From ubuntu
Run apt-get install ruby
Run echo done!

# After
From ubuntu
Run apt-get update
Run apt-get install ruby
Run echo done!
```

Since you add *Run apt-get update* instruction, **all** instructions after that have to be done from the scratch even if they are not changed. This is inevitable because Dockerfile uses the image built by the previous instruction as a parent image to execute next instruction. So, if you insert an instruction that creates a new parent image, all subsequent instructions cannot use cache because now parent image differs.

Cache is invalid even when adding commands that don't do anything

This invalidates caching. For example,

```
# Before
Run apt-get update

# After
Run apt-get update && true
```

Even if `true` command doesn't change anything of the image, Docker invalids the cache.

Cache is invalid when you add spaces between command and arguments inside instruction

This invalids cache

```
# Before
Run apt-get update

# After
Run apt-get               update
```

Cache is used when you add spaces around commands inside instruction

Cache is valid even if you add space around commands

```
# Before
Run apt-get update

# After
Run                apt-get update
```

Cache is used for non-idempotent instructions

This is kind of pitfall of build caching. What I mean by non-idempotent instructions is the execution of commands that may return different result each time. For example, `apt-get update` is not idempotent because the content of updates changes as time goes by.

```
From ubuntu
Run apt-get update
```

You made this Dockerfile and create image. 3 months later, Ubuntu made some security updates to their repository, so you rebuild the image by using the same Dockerfile hoping your new image includes the security updates. However, this doesn't pick up the updates. Since no instructions or files are changed, Docker uses cache and skips doing `apt-get update`.

If you don't want to use cache, just pass `-no-cache` option to build.

```
$ docker build -no-cache .
```

Instructions after ADD never cached (Only versions prior to 0.7.3)

If you use Docker before v7.3, watch out!

```
From ubuntu
Add myfile /
Run apt-get update
Run apt-get install openssh-server
```

If you have Dockerfile like this, ***Run apt-get update*** and ***Run apt-get install openssh-server*** will never be cached.

The behavior is changed from v7.3. It caches even if you have ADD instruction, but invalids cache if file content is changed.

```
$ echo "Jeff Beck" > rock.you

From ubuntu
Add rock.you /
Run add rock.you

$ echo "Eric Clapton" > rock.you

From ubuntu
Add rock.you /
Run add rock.you
```

Since you change *rock.you* file, instructions after Add doesn't use cache.

Hack to run container in the background

If you want to simplify the way to run containers, you should run your container on background with `docker run -d image your-command`. Instead of running with `docker run -i -t image your-command`, using `-d` is recommended because you can run your container with just one command and you don't need to detach terminal of container by hitting `Ctrl + P + Q`.

However, there is a problem with `-d` option. Your container immediately stops unless the commands are not running on foreground.

Let me explain this by using case where you want to run apache service on a container. The intuitive way of doing this is

```
$ docker run -d apache-server apachectl start
```

However, the container stops immediately after it is started. This is because `apachectl` exits once it detaches apache daemon.

Docker doesn't like this. Docker requires your command to keep running in the foreground. Otherwise, it thinks that your applications stops and shutdown the container.

You can solve this by directly running apache executable with foreground option.

```
$ docker run -e APACHE_RUN_USER=www-data \
                    -e APACHE_RUN_GROUP=www-data \
                    -e APACHE_PID_FILE=/var/run/apache2.pid \
                    -e APACHE_RUN_DIR=/var/run/apache2 \
                    -e APACHE_LOCK_DIR=/var/lock/apache2 \
                    -e APACHE_LOG_DIR=/var/log/apache2 \
                    -d apache-server /usr/sbin/apache2 -D NO_DETACH -D FOREGROUND
```

Here we are manually doing what `apachectl` does for us and run apache executable. With this approach, apache keeps running on foreground.

The problem is that some application does not run in the foreground. Also, we need to do extra works such as exporting environment variables by ourselves. How can we make it easier?

In this situation, you can add `tail -f /dev/null` to your command. By doing this, even if your main command runs in the background, your container doesn't stop because `tail` is keep running in the foreground. We can use this technique in the apache case.

```
$ docker run -d apache-server apachectl start && tail -f /dev/null
```

Much better, right? Since `tail -f /dev/null` doesn't do any harm, you can use this hack to any applications.

Comments

20 Comments Kim's Tech Blog 1 Login

Recommend 8 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Ionel Cristian Mărieș • 2 years ago

s/CND/CMD/

8 • Reply • Share ›

mafro • 3 years ago

This is super useful! Thanks! With regards to the first gotcha you mention, how does one achive adding /home/vagrant/myfile.txt?

1 • Reply • Share ›

Aaron Nel • 7 months ago

problem with using tail -f /dev/null is that you container, which would have stopped if there was an error that terminated a foreground application, now hangs around using up resources.

This has side effects that need to be carefully considered.

Better to put a wrapper script in place that will exit if your container service dies.
Fail fast, fail often and learn.

• Reply • Share ›

fieregurafiku • a year ago

As far as I can judge, the last command should be

> ```
> $ docker run -d 'apache-server apachectl start && tail -f /dev/null'
> ```

Quotation makes things different.

• Reply • Share ›

molind • 2 years ago

Thanks for article! Very useful.

• Reply • Share ›

catherinedevlin • 2 years ago

Thank you SO much - that "missing shebang" error on an ENTRYPOINT script is nasty!

• Reply • Share ›

Daniel Kahlenberg • 2 years ago

Very well explained, thank you for that article! I experienced the seemingly rather unintuitive features of the ADD directive myself earlier and your article helps to understand the specifics.

• Reply • Share ›

easyDonny • 2 years ago

The part about "invalids cache" is really helpful, thanks a lot!

• Reply • Share ›

lucaspottersky • 2 years ago

super hot advices! thanks!

• Reply • Share ›

superzamp • 3 years ago

Really helpful article, thanks !

• Reply • Share ›

OrHanSolo • 3 years ago

This has been an excellent reading, very useful indeed. Thanks.

• Reply • Share ›

Mani Agnihotri • 3 years ago

How can you make a command is always run and never reused from docker cache. My problem is

How can you make a command is always run and never reused from docker cache. My problem is --> RUN ./run.sh gets cached (because the file is downloaded from git). To avoid this I have to add --no-cache option to build the complete dockerfile which is one huge file and I don;t like the complete cache getting invalidated in it :(

• Reply • Share ›

kimhirokuni Mod ↪ Mani Agnihotri • 3 years ago

Hi, I don't think you can mix cached and non-cached commands in Dockerfile. If you want run.sh not to be cached and others to be cached, you have to run the shell script at exec time.

```
docker run -it you_image /bin/bash run.sh
```

• Reply • Share ›

jaya prakash ↪ kimhirokuni • a month ago

Newbie to docker , Can you explain me why you wrote /bin/bash in "docker run -it you_image /bin/bash run.sh ".....And also the difference between shell mode and exec mode..

• Reply • Share ›

Mani Agnihotri ↪ kimhirokuni • 3 years ago

Ok thanks. I have one more question. While building an image I want to expose a port to be accessible over the web. I have added the EXPOSE 8080 in my Dokerfile but i still am not able to hit the URL http://machine:8080. My dockerfile starts a server inside run.sh and then waits for 20 min before continuing. Is there someway to be able to map the port to machine port while building image from dockerfile? Like we can do while running a container -p 8080:8080

• Reply • Share ›

kimhirokuni Mod ↪ Mani Agnihotri • 3 years ago

Seems you can't really decide which host ports to map from within Dockerfile. See: https://docs.docker.com/ref...

It says: "EXPOSE doesn't define which ports can be exposed to the host or make ports accessible from the host by default. To expose ports to the host, at runtime, use the -p flag or the -P flag."

• Reply • Share ›

Keith Burns • 3 years ago

THANK YOU

• Reply • Share ›

Pawel Veselov • 3 years ago

One more strange gotcha : When ADDing files to a directory, one must add the trailing slash. Otherwise the error message is not reasonable.

ADD /myfile.txt /tmp/ # OK

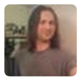
ADD /myfile.txt /tmp
# generates : stat
/var/lib/docker/devicemapper/mnt/c830705c46153463566fc01941d6377d9bbad80a0676e0884649a60e
not a directory

• Reply • Share ›

Ngalamo Leadel ➦ Pawel Veselov • 9 months ago

This does makes sense cuz
ADD /myfile.txt /tmp/ will add the file to a directory /tmp/
while ADD /myfile.txt /tmp will add the file to the root dir in a file call tmp which is not the case require i guess

• Reply • Share ›



Pawel Veselov ➦ Ngalamo Leadel • 9 months ago

/bin/cp disagrees.

• Reply • Share ›

Subscribe Add Disqus to your siteAdd DisqusAdd Privacy

[« Running Docker Containers Asynchronously with Celluloid](#) [Dockerfileを書く時の注意とかコツとかハックとか »](#)