Docker

Virtualization vs Containers

Virtualization and virtual machines

Virtualization is the process where a software called hypervisor (which sits on top of your native operating system) allows other operating systems to run on top of your native operating system.

The hypervisor creates virtual machines which is an emulation of physical computer, it creates virtual resources like CPU, RAM, disk, and networking giving the virtual machine the allusion that it is on its own independent computer. However, in reality it is actually managed by the hypervisor.

So with virtualization, it will recreate the underlying hardware for every virtual machines that you spin up. Which might be inefficient but it is good isolation as each of the operating system won't influence each other.

In addition, with virtual machines you are archiving isolation of machines. Each virtual machine are isolated from each other and you are able to run a full running operating system.

Underlying resources for each virtual machine are accessed by hypervisor.

Virtual machine you have flexibility on hardware, you can give say 5 virtual machines all 8 cores, despite only having 8 cores on your host machine.

Containers

Container is a lighter-weight more faster way of handling virtualization. They do not use a hypervisor but instead they use a container engine to run multiple container on the same kernel.

Container do not recreate the physical hardware and they package all the dependencies and software, and even the operating system itself that is required to execute the contained software application. This basically allows you to run your application anywhere, regardless of the physical hardware that's underneath.

Containers allows micro service architectures and they are kind of built for it.

With containers you are archiving isolation of processes. Each container will be run as a process and they are isolated from each other. Normal process that we run will able to inspect other processes! That is possible!!! However, with container when they run as a process, it will apparent to themselves that they don't see other processes being run, all the software and code that's necessary are packaged into them already.

Container's resources are accessed by kernel features.

Containers have portability, it has all the code that's necessary to run the container, you can take the docker image and run it pretty much anyway as long as you built it for ARM or x86 their corresponding architecture.

RUN, CMD, ENTRYPOINT Directives

RUN

The RUN directive is executed in a new layer, what does it mean? It is used to install packages and applications on top of an existing image layer and create a new layer on top of it. Docker images are used to build new ones and the RUN directive allows you to add more software / security updates on top of the based image, giving you the ability to customize it.

You have to understand that Docker images work in layers. You have the base images that are provided by Docker such as Ubuntu, Linux, or Windows images. They by default don't have much software installed, that's where you can use RUN to install other software to customize it to your need! By running RUN it will add a new layer on top of the existing layer with the software you want to install.

```
RUN [“apt-get”, “install”, “vim”]
```

This RUN command basically installs `vim` on top of your existing image which might not have it by default.

There can be multiple RUN command in your Dockerfile because it is used to customize the image to your need.

CMD

The CMD directive is similar to ENTRYPOINT both are used to run a command after your container has started. (Okay I got this environment setup already, with all the code and dependencies, what do you actually want me to run?).

Using the CMD command you set a default command. The default command is run if you run the container without specifying some command. In the case where you do specify command then the CMD line is ignored because the command user specifies override the default command.

> There can only be one CMD instruction in a Dockerfile, if you have more than one only the last CMD take effect.

There are three ways of using CMD directive

1. CMD <command> parameter1, parameter2... (Shell form)
2. CMD ["executable command", "parameter1", "parameter2"] (Executable form)
3. CMD ["parameter1", "parameter2"]

The third way is used to set additional default parameter when you are using ENTRYPOINT in executable form.

ENTRYPOINT

Very similar to CMD also used to tell what command to run after your container has started. Only the last ENTRYPOINT will have an effect.

However, if you use ENTRYPOINT command then you cannot override the instruction by adding command-line parameter to the `docker run` command. If you use ENTRYPOINT command then you are implying the container is built for a specific use-case and the command should not be overriden.

```
FROM ubuntu
ENTRYPOINT ["echo", "Hello World"]
```

If you have this as your Dockerfile and running

```
docker build -t entrypoint-instructions
docker run entrypoint-instructions
```

This will simply print out "Hello World". But what happens if you add command line arguments after the image name?

`docker run entrypoint-instructions printenv`

It will print out "Hello World printenv". So command-line arguments are simply appended as additional parameters to the ENTRYPOINT command.

If you do want to override ENTRYPOINT you would use the `--entrypoint` flag and then provide in the command you want to execute instead.

Combining ENTRYPOINT & CMD

If you have both ENTRYPOINT and CMD in your Dockerfile then CMD's parameter will just be appended to ENTRYPOINT as additional parameter just like what it would happen if you provide additional command-line arguments to ENTRYPOINT.

```
FROM ubuntu
ENTRYPOINT ["echo", "Hello"]
CMD ["Ricky"]
```

Running this container without any argument will print out "Hello Ricky"

However, if you run the container with argument such as

`docker run entrypoint-cmd hehexd`

Because you provided command-line argument it will override the CMD result in printing "Hello hehexd"

> When using both instructions use it in exec form.

exec form vs shell form

Command forms

`RUN, ENTRYPOINT, and CMD` are all directives to run a command in your Dockerfile. All three takes two forms of command, shell form and exec form.

1. Shell form
The commands are written without the `[]` brackets and are run by the container's shell as a child process, the normal way of running things in Linux and by default the shell is `/bin/sh -c`

```
FROM alphine:latest


# /bin/sh -c 'echo $HOME'
RUN echo $HOME


# /bin/sh -c 'echo $PATH
CMD echo $PATH
```

2. Exec form
The commands are written in `[]` brackets and are run directly, not through a shell, mean which means is the main process with PID 1. You will not get shell features like variable substitution from environment variable. In addition, you will not be able to pipe output, chain commands, IO redirections. Those features are only possible with shell form.

```
FROM alpine:latest


RUN ["pwd"]


CMD ["sleep", "1s"]
```

Differences between shell and exec form

In shell form, commands will inherit the environment variables from the shell

```
FROM alphine:latest


# This will echo out /root because it is run in the shell
RUN echo $HOME
```

```
# This will just echo "$HOME" because it isn't using variable substitution, a shell feature.
RUN ["echo", "$HOME"]
```

Signals

In addition, because the shell form runs the command via shell, i.e. it will spawn child process to run the commands if you are going to stop the container it will require 2 signals to stop it. In exec form because your process is the main process i.e. with PID of 1, when you stop the container docker only needs to send one signal to stop the main process.

The container stops when the main process PID of 1 stops.

In essence, shell form since the executable is not the main process, it will not receive signals.

In exec form since the executable is the main process (PID 1) it will receive signals.

Why does it take some time for container to stop

Docker when you stop a container will sent the main process (PID 1) a SIGTERM to ask it nicely to shutdown. If the main process doesn't handle SIGTERM like a shell script without `trap` then after a grace period (10 seconds), SIGKILL is sent to forcefully stop shutdown the process. Which is why if you run custom script that runs forever without handling SIGTERM it will take 10 seconds for it to be forcefully shutdown.

Recommended forms

- RUN: Use shell form
- ENTRYPOINT: Use exec form
- CMD: Use exec form

exec form vs shell form PT.2

shell form

The PID 1 is the shell, which will spawn the process that the program it is actually running.

Any environment variable referenced will be resolved to it's actual value.

```
ENTRYPOINT ./run.sh "$PATH"
```

If the shell script `run.sh` just echos out the parameter that it is passed in, then it will output the actual path variable.

Using shell form with exec form

If you have the above ENTRYPOINT in shell form and wanted to add exec form for CMD like below

```
ENTRYPOINT ./run.sh
CMD ["$PATH"]
```

This will NOT expand the environment variable, in fact it will NOT even print "$PATH". This is because shell form used in Dockerfile puts all the arguments in one big string.

So what's going on?

So in the previous example, `ENTRYPOINT ./run.sh "$PATH"` results in running the command below in the terminal

```
/bin/sh -c "./run.sh $PATH"
```

Since it is executed as part of the shell program, the environment variable will get expanded and passed into the shell script.

However, with the second example it actually results in running below in the terminal

```
/bin/sh -c "./run.sh" "$PATH"
```

The way the shell program works if you pass in the -c flag is that only the first parameter will be recognize as the program and the parameter you want to execute. Any parameter after it are just simply IGNORED! Which means $PATH isn't even passed into the shell script!

exec form

Exec form takes the form of array of strings separated by commas. The way how it works is that it uses the `exec` command underneath and it simply replaces the current running process with the one you have specified. In this case, any command running in exec form will have PID of 1.

And because it is simply replacement of running process, there is no shell involved, and thus there are not shell variable expansion.

Running:

```
ENTRYPOINT ["./run.sh", "$PATH"]
```

will just print out $PATH LITERALLY, because shell isn't involved to do any expansions.

Combining with CMD in shell form

If you combine it with CMD in shell form it will just print out /bin/sh -c with any parameters that you have included

Combining with CMD in exec form

If you combine it with CMD in exec form then it will just include any additional parameter you have specified, WITHOUT shell expansion.

docker stop vs docker kill

Docker stop

This provides a graceful way of exiting the container. it will sent a SIGTERM to the main process (PID 1), then after a grace period of 10 seconds if the main process still doesn't exit it will sent a SIGKILL to forcefully kill the main process.

Docker kill

By default, this command sends SIGKILL, so no graceful shutdown.

Why does my container immediately exits?

Containers are not like virtual machines

Each docker contain have a main process that is run via CMD / ENTRYPOINT command in the Dockerfile. Once those processes finishes and exit then the container will stop and exit as well!

It will not run indefinitely. Images like NGINX will run forever because it has a foreground process that is kept on listening for connections. If you take a look at the ENTRYPOINT command for NGINX image, it will run the nginx program with daemon off, so it is running in the foreground as the main process.

Images like php or ubuntu will exit immediately because the CMD that it executes by default is /bin/bash or php interpreters. And you didn't allocate an interactive shell via --interactive and -tty. Then it has no STDIN from the user, and therefore the /bin/bash just exits and thus container exits as well.

Therefore if you want to keep your php or ubuntu running, use -it and -d, if you aren't changing the default command, since the default command that is run is their interpreter.

For my use case I would change the CMD to start the PocketMine-MP server, and thus using -it isn't needed because the server will run indefinitely just like nginx. I would use -d to not see any of the outputs.

dumb-init for script

Problem

So you got your command that you would like to run it using a docker container, problem is once you got your program running using whatever mean possible, you see that it is running, but when you want the program to finish because it is say a web service, you notice that it takes a significant amount of time for the container to stop. You wonder to yourself, you read about `docker stop` sending sigterm then after a grace period of 10 seconds it will send sigkill to forcefully kill the process within the container.

You learned about that if the main program is ran under shell form, i.e. no JSON array, then your main process will be spawned by a shell, with that shell being PID of 1. Shell by itself doesn't handle sigterm signal, so you thought to yourself, let's make our main program the main process by switching to exec form.

Now you got your program running under exec form, and you want to stop it again, but there it is, it doesn't stop immediately either?! You know that your program is running under PID 1 so the sigterm signal should reach the main process, and it should gracefully stop because you tested locally. What's going on here?

Special PID 1

It turns out Linux kernel treats PID 1 as a special case, and applies different rules on how it handle signals. For a normal process if it doesn't register its own handlers for SIGTERM then it will use the default implementation for handling SIGTERM which is to kill the process.

However, for PID 1, kernel will NOT fallback to the default behavior. Therefore, SIGTERM will have no effect on the process. This is because PID 1 is run by the init process, and this is kind of safety mechanism to prevent people from accidentally sending SIGTERM to the init process.

This is why your program will not react to the SIGTERM signal, because it is treated as the "init" process by being PID of 1.

Resolution

To solve this, we would need an init process to take care of the signal forwarding and reaping of the process for us. Luckily we can do it in two ways:

1. Use the `--init` flag to use `Tini` as our init process
2. Use `#!/usr/bin/dumb-init` in your script to start a script with an init process that also takes care of the signal forwarding. Note that `dumb-init` needs to be installed in the image before it can be used because it is a separate program.