Estimated reading time: 61 minutes

Docker run reference

Docker runs processes in isolated containers. A container is a process which runs on a host. The host may be local or remote. When an operator executes `docker run`, the container process that runs is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host.

This page details how to use the `docker run` command to define the container’s resources at runtime.

General form

The basic `docker run` command takes this form:

```
$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

The `docker run` command must specify an *IMAGE* (https://docs.docker.com/engine/reference/glossary/#image) to derive the container from. An image developer can define image defaults related to:

- detached or foreground running
- container identification
- network settings
- runtime constraints on CPU and memory

With the `docker run [OPTIONS]` an operator can add to or override the image defaults set by a developer. And, additionally, operators can override nearly all the defaults set by the Docker runtime itself. The operator’s ability to override image and Docker runtime defaults is why *run* (https://docs.docker.com/engine/reference/commandline/run/) has more options than any other `docker` command.

To learn how to interpret the types of `[OPTIONS]`, see *Option types* (https://docs.docker.com/engine/reference/commandline/cli/#option-types).

> **Note**: Depending on your Docker system configuration, you may be required to preface the `docker run` command with `sudo`. To avoid having to use `sudo` with the `docker` command, your system administrator can create a Unix group called `docker` and add users to it. For more information about this configuration, refer to the Docker installation documentation for your operating system.

Operator exclusive options

Only the operator (the person executing `docker run`) can set the following options.

- Detached vs foreground (/engine/reference/run/#detached-vs-foreground)
  - Detached (-d) (/engine/reference/run/#detached--d)

    - Foreground (/engine/reference/run/#foreground)
- Container identification (/engine/reference/run/#container-identification)
    - Name (--name) (/engine/reference/run/#name---name)
    - PID equivalent (/engine/reference/run/#pid-equivalent)
- IPC settings (--ipc) (/engine/reference/run/#ipc-settings---ipc)
- Network settings (/engine/reference/run/#network-settings)
- Restart policies (--restart) (/engine/reference/run/#restart-policies---restart)
- Clean up (--rm) (/engine/reference/run/#clean-up---rm)
- Runtime constraints on resources (/engine/reference/run/#runtime-constraints-on-resources)
- Runtime privilege and Linux capabilities (/engine/reference/run/#runtime-privilege-and-linux-capabilities)

Detached vs foreground

When starting a Docker container, you must first decide if you want to run the container in the background in a "detached" mode or in the default foreground mode:

```
-d=false: Detached mode: Run container in the background, print new container id
```

Detached (-d)

To start a container in detached mode, you use `-d=true` or just `-d` option. By design, containers started in detached mode exit when the root process used to run the container exits, unless you also specify the `--rm` option. If you use `-d` with `--rm`, the container is removed when it exits **or** when the daemon exits, whichever happens first.

Do not pass a `service x start` command to a detached container. For example, this command attempts to start the `nginx` service.

```
$ docker run -d -p 80:80 my_image service nginx start
```

This succeeds in starting the `nginx` service inside the container. However, it fails the detached container paradigm in that, the root process (`service nginx start`) returns and the detached container stops as designed. As a result, the `nginx` service is started but could not be used. Instead, to start a process such as the `nginx` web server do the following:

```
$ docker run -d -p 80:80 my_image nginx -g 'daemon off;'
```

To do input/output with a detached container use network connections or shared volumes. These are required because the container is no longer listening to the command line where `docker run` was run.

To reattach to a detached container, use `docker` *attach* (https://docs.docker.com/engine/reference/commandline/attach/) command.

Foreground

In foreground mode (the default when `-d` is not specified), `docker run` can start the process in the container and attach the console to the process's standard input, output, and standard error. It can even pretend to be a TTY (this is what most command line executables expect) and pass along signals. All of that is configurable:

```
-a=[]           : Attach to `STDIN`, `STDOUT` and/or `STDERR`
-t              : Allocate a pseudo-tty
--sig-proxy=true: Proxy all received signals to the process (non-TTY mode only)
-i              : Keep STDIN open even if not attached
```

If you do not specify `-a` then Docker will attach to both stdout and stderr (https://github.com/docker/docker/blob/4118e0c9eebda2412a09ae66e90c34b85fae3275/runconfig/opts/parse.go#L267). You can specify to which of the three standard streams (`STDIN` , `STDOUT` , `STDERR`) you'd like to connect instead, as in:

```
$ docker run -a stdin -a stdout -i -t ubuntu /bin/bash
```

For interactive processes (like a shell), you must use `-i -t` together in order to allocate a tty for the container process. `-i -t` is often written `-it` as you'll see in later examples. Specifying `-t` is forbidden when the client is receiving its standard input from a pipe, as in:

```
$ echo test | docker run -i busybox cat
```

> **Note**: A process running as PID 1 inside a container is treated specially by Linux: it ignores any signal with the default action. So, the process will not terminate on `SIGINT` or `SIGTERM` unless it is coded to do so.

Container identification

Name (--name)

The operator can identify a container in three ways:

Identifier type	Example value
UUID long identifier	"f78375b1c487e03c9438c729345e54db9d20cfa2ac1fc3494b6eb60872e74778"
UUID short identifier	"f78375b1c487"
Name	"evil_ptolemy"

The UUID identifiers come from the Docker daemon. If you do not assign a container name with the `--name` option, then the daemon generates a random string name for you. Defining a `name` can be a handy way to add meaning to a container. If you specify a `name` , you can use it when referencing the container within a

Docker network. This works for both background and foreground Docker containers.

> **Note**: Containers on the default bridge network must be linked to communicate by name.

PID equivalent

Finally, to help with automation, you can have Docker write the container ID out to a file of your choosing. This is similar to how some programs might write out their process ID to a file (you've seen them as PID files):

```
--cidfile="": Write the container ID to the file
```

Image[:tag]

While not strictly a means of identifying a container, you can specify a version of an image you'd like to run the container with by adding `image[:tag]` to the command. For example, `docker run ubuntu:14.04` .

Image[@digest]

Images using the v2 or later image format have a content-addressable identifier called a digest. As long as the input used to generate the image is unchanged, the digest value is predictable and referenceable.

The following example runs a container from the `alpine` image with the `sha256:9cacb71397b640eca97488cf08582ae4e4068513101088e9f96c9814bfda95e0` digest:

```
$ docker run alpine@sha256:9cacb71397b640eca97488cf08582ae4e4068513101088e9f96c9814bfda95e0 date
```

PID settings (--pid)

```
--pid=""  : Set the PID (Process) Namespace mode for the container,
             'container:<name|id>': joins another container's PID namespace
             'host': use the host's PID namespace inside the container
```

By default, all containers have the PID namespace enabled.

PID namespace provides separation of processes. The PID Namespace removes the view of the system processes, and allows process ids to be reused including pid 1.

In certain cases you want your container to share the host's process namespace, basically allowing processes within the container to see all of the processes on the system. For example, you could build a container with debugging tools like `strace` or `gdb` , but want to use these tools when debugging processes within the container.

Example: run htop inside a container

Create this Dockerfile:

```
FROM alpine:latest
RUN apk add --update htop && rm -rf /var/cache/apk/*
CMD ["htop"]
```

Build the Dockerfile and tag the image as `myhtop`:

```
$ docker build -t myhtop .
```

Use the following command to run `htop` inside a container:

```
$ docker run -it --rm --pid=host myhtop
```

Joining another container’s pid namespace can be used for debugging that container.

Example

Start a container running a redis server:

```
$ docker run --name my-redis -d redis
```

Debug the redis container by running another container that has strace in it:

```
$ docker run -it --pid=container:my-redis my_strace_docker_image bash
$ strace -p 1
```

UTS settings (--uts)

```
--uts=""  : Set the UTS namespace mode for the container,
       'host': use the host's UTS namespace inside the container
```

The UTS namespace is for setting the hostname and the domain that is visible to running processes in that namespace. By default, all containers, including those with `--network=host`, have their own UTS namespace. The `host` setting will result in the container using the same UTS namespace as the host. Note that `--hostname` is invalid in `host` UTS mode.

You may wish to share the UTS namespace with the host if you would like the hostname of the container to change as the hostname of the host changes. A more advanced use case would be changing the host’s hostname from a container.

IPC settings (--ipc)

```
--ipc="MODE"  : Set the IPC mode for the container
```

The following values are accepted:

Value	Description
""	Use daemon's default.
"none"	Own private IPC namespace, with /dev/shm not mounted.
"private"	Own private IPC namespace.
"shareable"	Own private IPC namespace, with a possibility to share it with other containers.
"container: <_name-or-ID_>"	Join another ("shareable") container's IPC namespace.
"host"	Use the host system's IPC namespace.

If not specified, daemon default is used, which can either be `"private"` or `"shareable"` , depending on the daemon version and configuration.

IPC (POSIX/SysV IPC) namespace provides separation of named shared memory segments, semaphores and message queues.

Shared memory segments are used to accelerate inter-process communication at memory speed, rather than through pipes or through the network stack. Shared memory is commonly used by databases and custom-built (typically C/OpenMPI, C++/using boost libraries) high performance applications for scientific computing and financial services industries. If these types of applications are broken into multiple containers, you might need to share the IPC mechanisms of the containers, using `"shareable"` mode for the main (i.e. "donor") container, and `"container:<donor-name-or-ID>"` for other containers.

Network settings

```
--dns=[]           : Set custom dns servers for the container
--network="bridge" : Connect a container to a network
                      'bridge': create a network stack on the default Docker bridge
                      'none': no networking
                      'container:<name|id>': reuse another container's network stack
                      'host': use the Docker host network stack
                      '<network-name>|<network-id>': connect to a user-defined network
--network-alias=[] : Add network-scoped alias for the container
--add-host=""      : Add a line to /etc/hosts (host:IP)
--mac-address=""   : Sets the container's Ethernet device's MAC address
--ip=""            : Sets the container's Ethernet device's IPv4 address
--ip6=""           : Sets the container's Ethernet device's IPv6 address
--link-local-ip=[] : Sets one or more container's Ethernet device's link local IPv4/IPv6 addresses
```

By default, all containers have networking enabled and they can make any outgoing connections. The operator can completely disable networking with `docker run --network none` which disables all incoming and outgoing networking. In cases like this, you would perform I/O through files or `STDIN` and `STDOUT` only.

Publishing ports and linking to other containers only works with the default (bridge). The linking feature is a legacy feature. You should always prefer using Docker network drivers over linking.

Your container will use the same DNS servers as the host by default, but you can override this with `--dns` .

By default, the MAC address is generated using the IP address allocated to the container. You can set the container's MAC address explicitly by providing a MAC address via the `--mac-address` parameter (format: `12:34:56:78:9a:bc`).Be aware that Docker does not check if manually specified MAC addresses are unique.

Supported networks :

Network	Description
none	No networking in the container.
bridge (default)	Connect the container to the bridge via veth interfaces.
host	Use the host's network stack inside the container.
container: <name|id>	Use the network stack of another container, specified via its *name* or *id*.
NETWORK	Connects the container to a user created network (using `docker network create` command)

NETWORK: NONE

With the network is `none` a container will not have access to any external routes. The container will still have a `loopback` interface enabled in the container but it does not have any routes to external traffic.

NETWORK: BRIDGE

With the network set to `bridge` a container will use docker's default networking setup. A bridge is setup on the host, commonly named `docker0` , and a pair of `veth` interfaces will be created for the container. One side of the `veth` pair will remain on the host attached to the bridge while the other side of the pair will be placed inside the container's namespaces in addition to the `loopback` interface. An IP address will be allocated for containers on the bridge's network and traffic will be routed though this bridge to the container.

Containers can communicate via their IP addresses by default. To communicate by name, they must be linked.

NETWORK: HOST

With the network set to `host` a container will share the host's network stack and all interfaces from the host will be available to the container. The container's hostname will match the hostname on the host system. Note that `--mac-address` is invalid in `host` netmode. Even in `host` network mode a container has its own UTS namespace by default. As such `--hostname` is allowed in `host` network mode and will only change the hostname inside the container. Similar to `--hostname` , the `--add-host` , `--dns` , `--dns-search` , and

`--dns-option` options can be used in `host` network mode. These options update `/etc/hosts` or `/etc/resolv.conf` inside the container. No change are made to `/etc/hosts` and `/etc/resolv.conf` on the host.

Compared to the default `bridge` mode, the `host` mode gives *significantly* better networking performance since it uses the host's native networking stack whereas the bridge has to go through one level of virtualization through the docker daemon. It is recommended to run containers in this mode when their networking performance is critical, for example, a production Load Balancer or a High Performance Web Server.

> **Note**: `--network="host"` gives the container full access to local system services such as D-bus and is therefore considered insecure.

NETWORK: CONTAINER

With the network set to `container` a container will share the network stack of another container. The other container's name must be provided in the format of `--network container:<name|id>` . Note that `--add-host` `--hostname` `--dns` `--dns-search` `--dns-option` and `--mac-address` are invalid in `container` netmode, and `--publish` `--publish-all` `--expose` are also invalid in `container` netmode.

Example running a Redis container with Redis binding to `localhost` then running the `redis-cli` command and connecting to the Redis server over the `localhost` interface.

```
$ docker run -d --name redis example/redis --bind 127.0.0.1
$ # use the redis container's network stack to access localhost
$ docker run --rm -it --network container:redis example/redis-cli -h 127.0.0.1
```

USER-DEFINED NETWORK

You can create a network using a Docker network driver or an external network driver plugin. You can connect multiple containers to the same network. Once connected to a user-defined network, the containers can communicate easily using only another container's IP address or name.

For `overlay` networks or custom plugins that support multi-host connectivity, containers connected to the same multi-host network but launched from different Engines can also communicate in this way.

The following example creates a network using the built-in `bridge` network driver and running a container in the created network

```
$ docker network create -d bridge my-net
$ docker run --network=my-net -itd --name=container3 busybox
```

Managing /etc/hosts

Your container will have lines in `/etc/hosts` which define the hostname of the container itself as well as `localhost` and a few other common things. The `--add-host` flag can be used to add additional lines to `/etc/hosts` .

```
$ docker run -it --add-host db-static:86.75.30.9 ubuntu cat /etc/hosts
172.17.0.22     09d03f76bf2c
fe00::0         ip6-localnet
ff00::0         ip6-mcastprefix
ff02::1         ip6-allnodes
ff02::2         ip6-allrouters
127.0.0.1       localhost
::1	            localhost ip6-localhost ip6-loopback
86.75.30.9      db-static
```

If a container is connected to the default bridge network and `linked` with other containers, then the container's `/etc/hosts` file is updated with the linked container's name.

> **Note** Since Docker may live update the container's `/etc/hosts` file, there may be situations when processes inside the container can end up reading an empty or incomplete `/etc/hosts` file. In most cases, retrying the read again should fix the problem.

Restart policies (--restart)

Using the `--restart` flag on Docker run you can specify a restart policy for how a container should or should not be restarted on exit.

When a restart policy is active on a container, it will be shown as either `Up` or `Restarting` in `docker ps` (https://docs.docker.com/engine/reference/commandline/ps/). It can also be useful to use `docker events` (https://docs.docker.com/engine/reference/commandline/events/) to see the restart policy in effect.

Docker supports the following restart policies:

Policy	Result
no	Do not automatically restart the container when it exits. This is the default.
on-failure[:max-retries]	Restart only if the container exits with a non-zero exit status. Optionally, limit the number of restart retries the Docker daemon attempts.
always	Always restart the container regardless of the exit status. When you specify always, the Docker daemon will try to restart the container indefinitely. The container will also always start on daemon startup, regardless of the current state of the container.
unless-stopped	Always restart the container regardless of the exit status, including on daemon startup, except if the container was put into a stopped state before the Docker daemon was stopped.

An ever increasing delay (double the previous delay, starting at 100 milliseconds) is added before each restart to prevent flooding the server. This means the daemon will wait for 100 ms, then 200 ms, 400, 800, 1600, and so on until either the `on-failure` limit is hit, or when you `docker stop` or `docker rm -f` the container.

If a container is successfully restarted (the container is started and runs for at least 10 seconds), the delay is reset to its default value of 100 ms.

You can specify the maximum amount of times Docker will try to restart the container when using the **on-failure** policy. The default is that Docker will try forever to restart the container. The number of (attempted) restarts for a container can be obtained via `docker inspect` (https://docs.docker.com/engine/reference/commandline/inspect/). For example, to get the number of restarts for container "my-container";

```
$ docker inspect -f "{{ .RestartCount }}" my-container
# 2
```

Or, to get the last time the container was (re)started;

```
$ docker inspect -f "{{ .State.StartedAt }}" my-container
# 2015-03-04T23:47:07.691840179Z
```

Combining `--restart` (restart policy) with the `--rm` (clean up) flag results in an error. On container restart, attached clients are disconnected. See the examples on using the `--rm` (clean up) (/engine/reference/run/#clean-up-rm) flag later in this page.

Examples

```
$ docker run --restart=always redis
```

This will run the `redis` container with a restart policy of **always** so that if the container exits, Docker will restart it.

```
$ docker run --restart=on-failure:10 redis
```

This will run the `redis` container with a restart policy of **on-failure** and a maximum restart count of 10. If the `redis` container exits with a non-zero exit status more than 10 times in a row Docker will abort trying to restart the container. Providing a maximum restart limit is only valid for the **on-failure** policy.

Exit Status

The exit code from `docker run` gives information about why the container failed to run or why it exited. When `docker run` exits with a non-zero code, the exit codes follow the `chroot` standard, see below:

125 if the error is with Docker daemon ***itself***

```
$ docker run --foo busybox; echo $?
# flag provided but not defined: --foo
  See 'docker run --help'.
  125
```

126 if the ***contained command*** cannot be invoked

```
$ docker run busybox /etc; echo $?
# docker: Error response from daemon: Container command '/etc' could not be invoked.
  126
```

127 if the ***contained command*** cannot be found

```
$ docker run busybox foo; echo $?
# docker: Error response from daemon: Container command 'foo' not found or does not exist.
  127
```

Exit code of ***contained command*** otherwise

```
$ docker run busybox /bin/sh -c 'exit 3'; echo $?
# 3
```

Clean up (--rm)

By default a container's file system persists even after the container exits. This makes debugging a lot easier (since you can inspect the final state) and you retain all your data by default. But if you are running short-term **foreground** processes, these container file systems can really pile up. If instead you'd like Docker to **automatically clean up the container and remove the file system when the container exits**, you can add the `--rm` flag:

```
--rm=false: Automatically remove the container when it exits
```

> **Note**: When you set the `--rm` flag, Docker also removes the anonymous volumes associated with the container when the container is removed. This is similar to running `docker rm -v my-container`. Only volumes that are specified without a name are removed. For example, with `docker run --rm -v /foo -v awesome:/bar busybox top`, the volume for `/foo` will be removed, but the volume for `/bar` will not. Volumes inherited via `--volumes-from` will be removed with the same logic -- if the original volume was specified with a name it will **not** be removed.

Security configuration

```
--security-opt="label=user:USER"     : Set the label user for the container
--security-opt="label=role:ROLE"     : Set the label role for the container
--security-opt="label=type:TYPE"     : Set the label type for the container
--security-opt="label=level:LEVEL"   : Set the label level for the container
--security-opt="label=disable"       : Turn off label confinement for the container
--security-opt="apparmor=PROFILE"    : Set the apparmor profile to be applied to the container
--security-opt="no-new-privileges:true|false"   : Disable/enable container processes from gaining
--security-opt="seccomp=unconfined"  : Turn off seccomp confinement for the container
--security-opt="seccomp=profile.json": White listed syscalls seccomp Json file to be used as a sec
```

You can override the default labeling scheme for each container by specifying the `--security-opt` flag. Specifying the level in the following command allows you to share the same content between containers.

```
$ docker run --security-opt label=level:s0:c100,c200 -it fedora bash
```

Note: Automatic translation of MLS labels is not currently supported.

To disable the security labeling for this container versus running with the `--privileged` flag, use the following command:

```
$ docker run --security-opt label=disable -it fedora bash
```

If you want a tighter security policy on the processes within a container, you can specify an alternate type for the container. You could run a container that is only allowed to listen on Apache ports by executing the following command:

```
$ docker run --security-opt label=type:svirt_apache_t -it centos bash
```

Note: You would have to write policy defining a `svirt_apache_t` type.

If you want to prevent your container processes from gaining additional privileges, you can execute the following command:

```
$ docker run --security-opt no-new-privileges -it centos bash
```

This means that commands that raise privileges such as `su` or `sudo` will no longer work. It also causes any seccomp filters to be applied later, after privileges have been dropped which may mean you can have a more restrictive set of filters. For more details, see the kernel documentation (https://www.kernel.org/doc/Documentation/prctl/no_new_privs.txt).

Specify an init process

You can use the `--init` flag to indicate that an init process should be used as the PID 1 in the container. Specifying an init process ensures the usual responsibilities of an init system, such as reaping zombie processes, are performed inside the created container.

The default init process used is the first `docker-init` executable found in the system path of the Docker daemon process. This `docker-init` binary, included in the default installation, is backed by tini (https://github.com/krallin/tini).

Specify custom cgroups

Using the `--cgroup-parent` flag, you can pass a specific cgroup to run a container in. This allows you to create and manage cgroups on their own. You can define custom resources for those cgroups and put containers under a common parent group.

Runtime constraints on resources

The operator can also adjust the performance parameters of the container:

Option	Description
`-m` , `--memory=""`	Memory limit (format: `<number>[<unit>]`). Number is a positive integer. Unit can be one of `b` , `k` , `m` , or `g` . Minimum is 4M.
`--memory-swap=""`	Total memory limit (memory + swap, format: `<number>[<unit>]`). Number is a positive integer. Unit can be one of `b` , `k` , `m` , or `g` .
`--memory-reservation=""`	Memory soft limit (format: `<number>[<unit>]`). Number is a positive integer. Unit can be one of `b` , `k` , `m` , or `g` .
`--kernel-memory=""`	Kernel memory limit (format: `<number>[<unit>]`). Number is a positive integer. Unit can be one of `b` , `k` , `m` , or `g` . Minimum is 4M.
`-c` , `--cpu-shares=0`	CPU shares (relative weight)
`--cpus=0.000`	Number of CPUs. Number is a fractional number. 0.000 means no limit.
`--cpu-period=0`	Limit the CPU CFS (Completely Fair Scheduler) period
`--cpuset-cpus=""`	CPUs in which to allow execution (0-3, 0,1)
`--cpuset-mems=""`	Memory nodes (MEMs) in which to allow execution (0-3, 0,1). Only effective on NUMA systems.
`--cpu-quota=0`	Limit the CPU CFS (Completely Fair Scheduler) quota
`--cpu-rt-period=0`	Limit the CPU real-time period. In microseconds. Requires parent cgroups be set and cannot be higher than parent. Also check rtprio ulimits.
`--cpu-rt-runtime=0`	Limit the CPU real-time runtime. In microseconds. Requires parent cgroups be set and cannot be higher than parent. Also check rtprio ulimits.

Option	Description
`--blkio-weight=0`	Block IO weight (relative weight) accepts a weight value between 10 and 1000.
`--blkio-weight-device=""`	Block IO weight (relative device weight, format: `DEVICE_NAME:WEIGHT`)
`--device-read-bps=""`	Limit read rate from a device (format: `<device-path>:<number>[<unit>]`). Number is a positive integer. Unit can be one of `kb` , `mb` , or `gb` .
`--device-write-bps=""`	Limit write rate to a device (format: `<device-path>:<number>[<unit>]`). Number is a positive integer. Unit can be one of `kb` , `mb` , or `gb` .
`--device-read-iops=""`	Limit read rate (IO per second) from a device (format: `<device-path>:<number>`). Number is a positive integer.
`--device-write-iops=""`	Limit write rate (IO per second) to a device (format: `<device-path>:<number>`). Number is a positive integer.
`--oom-kill-disable=false`	Whether to disable OOM Killer for the container or not.
`--oom-score-adj=0`	Tune container’s OOM preferences (-1000 to 1000)
`--memory-swappiness=""`	Tune a container’s memory swappiness behavior. Accepts an integer between 0 and 100.
`--shm-size=""`	Size of `/dev/shm` . The format is `<number><unit>` . `number` must be greater than `0` . Unit is optional and can be `b` (bytes), `k` (kilobytes), `m` (megabytes), or `g` (gigabytes). If you omit the unit, the system uses bytes. If you omit the size entirely, the system uses `64m` .

User memory constraints

We have four ways to set user memory usage:

Option	Result
memory=inf, memory-swap=inf (default)	There is no memory limit for the container. The container can use as much memory as needed.
memory=L<inf, memory-swap=inf	(specify memory and set memory-swap as `-1`) The container is not allowed to use more than L bytes of memory, but can use as much swap as is needed (if the host supports swap memory).
memory=L<inf, memory-swap=2*L	(specify memory without memory-swap) The container is not allowed to use more than L bytes of memory, swap *plus* memory usage is double of that.
memory=L<inf, memory-swap=S<inf, L<=S	(specify both memory and memory-swap) The container is not allowed to use more than L bytes of memory, swap *plus* memory usage is limited by S.

Examples:

```
$ docker run -it ubuntu:14.04 /bin/bash
```

We set nothing about memory, this means the processes in the container can use as much memory and swap memory as they need.

```
$ docker run -it -m 300M --memory-swap -1 ubuntu:14.04 /bin/bash
```

We set memory limit and disabled swap memory limit, this means the processes in the container can use 300M memory and as much swap memory as they need (if the host supports swap memory).

```
$ docker run -it -m 300M ubuntu:14.04 /bin/bash
```

We set memory limit only, this means the processes in the container can use 300M memory and 300M swap memory, by default, the total virtual memory size (--memory-swap) will be set as double of memory, in this case, memory + swap would be 2*300M, so processes can use 300M swap memory as well.

```
$ docker run -it -m 300M --memory-swap 1G ubuntu:14.04 /bin/bash
```

We set both memory and swap memory, so the processes in the container can use 300M memory and 700M swap memory.

Memory reservation is a kind of memory soft limit that allows for greater sharing of memory. Under normal circumstances, containers can use as much of the memory as needed and are constrained only by the hard limits set with the `-m` / `--memory` option. When memory reservation is set, Docker detects memory contention or low memory and forces containers to restrict their consumption to a reservation limit.

Always set the memory reservation value below the hard limit, otherwise the hard limit takes precedence. A reservation of 0 is the same as setting no reservation. By default (without reservation set), memory reservation is the same as the hard memory limit.

Memory reservation is a soft-limit feature and does not guarantee the limit won't be exceeded. Instead, the feature attempts to ensure that, when memory is heavily contended for, memory is allocated based on the reservation hints/setup.

The following example limits the memory (`-m`) to 500M and sets the memory reservation to 200M.

```
$ docker run -it -m 500M --memory-reservation 200M ubuntu:14.04 /bin/bash
```

Under this configuration, when the container consumes memory more than 200M and less than 500M, the next system memory reclaim attempts to shrink container memory below 200M.

The following example set memory reservation to 1G without a hard memory limit.

```
$ docker run -it --memory-reservation 1G ubuntu:14.04 /bin/bash
```

The container can use as much memory as it needs. The memory reservation setting ensures the container doesn't consume too much memory for long time, because every memory reclaim shrinks the container's consumption to the reservation.

By default, kernel kills processes in a container if an out-of-memory (OOM) error occurs. To change this behaviour, use the `--oom-kill-disable` option. Only disable the OOM killer on containers where you have also set the `-m/--memory` option. If the `-m` flag is not set, this can result in the host running out of memory and require killing the host's system processes to free memory.

The following example limits the memory to 100M and disables the OOM killer for this container:

```
$ docker run -it -m 100M --oom-kill-disable ubuntu:14.04 /bin/bash
```

The following example, illustrates a dangerous way to use the flag:

```
$ docker run -it --oom-kill-disable ubuntu:14.04 /bin/bash
```

The container has unlimited memory which can cause the host to run out memory and require killing system processes to free memory. The `--oom-score-adj` parameter can be changed to select the priority of which containers will be killed when the system is out of memory, with negative scores making them less likely to be killed, and positive scores more likely.

Kernel memory constraints

Kernel memory is fundamentally different than user memory as kernel memory can't be swapped out. The inability to swap makes it possible for the container to block system services by consuming too much kernel memory. Kernel memory includes：

- stack pages
- slab pages
- sockets memory pressure
- tcp memory pressure

You can setup kernel memory limit to constrain these kinds of memory. For example, every process consumes some stack pages. By limiting kernel memory, you can prevent new processes from being created when the kernel memory usage is too high.

Kernel memory is never completely independent of user memory. Instead, you limit kernel memory in the context of the user memory limit. Assume "U" is the user memory limit and "K" the kernel limit. There are three possible ways to set limits:

Option	Result
U != 0, K = inf (default)	This is the standard memory limitation mechanism already present before using kernel memory. Kernel memory is completely ignored.

Option	Result
U != 0, K < U	Kernel memory is a subset of the user memory. This setup is useful in deployments where the total amount of memory per-cgroup is overcommitted. Overcommitting kernel memory limits is definitely not recommended, since the box can still run out of non-reclaimable memory. In this case, you can configure K so that the sum of all groups is never greater than the total memory. Then, freely set U at the expense of the system's service quality.
U != 0, K > U	Since kernel memory charges are also fed to the user counter and reclamation is triggered for the container for both kinds of memory. This configuration gives the admin a unified view of memory. It is also useful for people who just want to track kernel memory usage.

Examples:

```
$ docker run -it -m 500M --kernel-memory 50M ubuntu:14.04 /bin/bash
```

We set memory and kernel memory, so the processes in the container can use 500M memory in total, in this 500M memory, it can be 50M kernel memory tops.

```
$ docker run -it --kernel-memory 50M ubuntu:14.04 /bin/bash
```

We set kernel memory without **-m**, so the processes in the container can use as much memory as they want, but they can only use 50M kernel memory.

Swappiness constraint

By default, a container’s kernel can swap out a percentage of anonymous pages. To set this percentage for a container, specify a `--memory-swappiness` value between 0 and 100. A value of 0 turns off anonymous page swapping. A value of 100 sets all anonymous pages as swappable. By default, if you are not using `--memory-swappiness` , memory swappiness value will be inherited from the parent.

For example, you can set:

```
$ docker run -it --memory-swappiness=0 ubuntu:14.04 /bin/bash
```

Setting the `--memory-swappiness` option is helpful when you want to retain the container’s working set and to avoid swapping performance penalties.

CPU share constraint

By default, all containers get the same proportion of CPU cycles. This proportion can be modified by changing the container’s CPU share weighting relative to the weighting of all other running containers.

To modify the proportion from the default of 1024, use the `-c` or `--cpu-shares` flag to set the weighting to 2 or higher. If 0 is set, the system will ignore the value and use the default of 1024.

The proportion will only apply when CPU-intensive processes are running. When tasks in one container are idle, other containers can use the left-over CPU time. The actual amount of CPU time will vary depending on the number of containers running on the system.

For example, consider three containers, one has a cpu-share of 1024 and two others have a cpu-share setting of 512. When processes in all three containers attempt to use 100% of CPU, the first container would receive 50% of the total CPU time. If you add a fourth container with a cpu-share of 1024, the first container only gets 33% of the CPU. The remaining containers receive 16.5%, 16.5% and 33% of the CPU.

On a multi-core system, the shares of CPU time are distributed over all CPU cores. Even if a container is limited to less than 100% of CPU time, it can use 100% of each individual CPU core.

For example, consider a system with more than three cores. If you start one container `{C0}` with `-c=512` running one process, and another container `{C1}` with `-c=1024` running two processes, this can result in the following division of CPU shares:

```
PID    container        CPU     CPU share
100    {C0}             0       100% of CPU0
101    {C1}             1       100% of CPU1
102    {C1}             2       100% of CPU2
```

CPU period constraint

The default CPU CFS (Completely Fair Scheduler) period is 100ms. We can use `--cpu-period` to set the period of CPUs to limit the container's CPU usage. And usually `--cpu-period` should work with `--cpu-quota` .

Examples:

```
$ docker run -it --cpu-period=50000 --cpu-quota=25000 ubuntu:14.04 /bin/bash
```

If there is 1 CPU, this means the container can get 50% CPU worth of run-time every 50ms.

In addition to use `--cpu-period` and `--cpu-quota` for setting CPU period constraints, it is possible to specify `--cpus` with a float number to achieve the same purpose. For example, if there is 1 CPU, then `--cpus=0.5` will achieve the same result as setting `--cpu-period=50000` and `--cpu-quota=25000` (50% CPU).

The default value for `--cpus` is `0.000` , which means there is no limit.

For more information, see the CFS documentation on bandwidth limiting (https://www.kernel.org/doc/Documentation/scheduler/sched-bwc.txt).

Cpuset constraint

We can set cpus in which to allow execution for containers.

Examples:

```
$ docker run -it --cpuset-cpus="1,3" ubuntu:14.04 /bin/bash
```

This means processes in container can be executed on cpu 1 and cpu 3.

```
$ docker run -it --cpuset-cpus="0-2" ubuntu:14.04 /bin/bash
```

This means processes in container can be executed on cpu 0, cpu 1 and cpu 2.

We can set mems in which to allow execution for containers. Only effective on NUMA systems.

Examples:

```
$ docker run -it --cpuset-mems="1,3" ubuntu:14.04 /bin/bash
```

This example restricts the processes in the container to only use memory from memory nodes 1 and 3.

```
$ docker run -it --cpuset-mems="0-2" ubuntu:14.04 /bin/bash
```

This example restricts the processes in the container to only use memory from memory nodes 0, 1 and 2.

CPU quota constraint

The `--cpu-quota` flag limits the container's CPU usage. The default 0 value allows the container to take 100% of a CPU resource (1 CPU). The CFS (Completely Fair Scheduler) handles resource allocation for executing processes and is default Linux Scheduler used by the kernel. Set this value to 50000 to limit the container to 50% of a CPU resource. For multiple CPUs, adjust the `--cpu-quota` as necessary. For more information, see the CFS documentation on bandwidth limiting (https://www.kernel.org/doc/Documentation/scheduler/sched-bwc.txt).

Block IO bandwidth (Blkio) constraint

By default, all containers get the same proportion of block IO bandwidth (blkio). This proportion is 500. To modify this proportion, change the container's blkio weight relative to the weighting of all other running containers using the `--blkio-weight` flag.

> **Note:** The blkio weight setting is only available for direct IO. Buffered IO is not currently supported.

The `--blkio-weight` flag can set the weighting to a value between 10 to 1000. For example, the commands below create two containers with different blkio weight:

```
$ docker run -it --name c1 --blkio-weight 300 ubuntu:14.04 /bin/bash
$ docker run -it --name c2 --blkio-weight 600 ubuntu:14.04 /bin/bash
```

If you do block IO in the two containers at the same time, by, for example:

```
$ time dd if=/mnt/zerofile of=test.out bs=1M count=1024 oflag=direct
```

You'll find that the proportion of time is the same as the proportion of blkio weights of the two containers.

The `--blkio-weight-device="DEVICE_NAME:WEIGHT"` flag sets a specific device weight. The `DEVICE_NAME:WEIGHT` is a string containing a colon-separated device name and weight. For example, to set `/dev/sda` device weight to `200` :

```
$ docker run -it \
    --blkio-weight-device "/dev/sda:200" \
    ubuntu
```

If you specify both the `--blkio-weight` and `--blkio-weight-device` , Docker uses the `--blkio-weight` as the default weight and uses `--blkio-weight-device` to override this default with a new value on a specific device. The following example uses a default weight of `300` and overrides this default on `/dev/sda` setting that weight to `200` :

```
$ docker run -it \
    --blkio-weight 300 \
    --blkio-weight-device "/dev/sda:200" \
    ubuntu
```

The `--device-read-bps` flag limits the read rate (bytes per second) from a device. For example, this command creates a container and limits the read rate to `1mb` per second from `/dev/sda` :

```
$ docker run -it --device-read-bps /dev/sda:1mb ubuntu
```

The `--device-write-bps` flag limits the write rate (bytes per second) to a device. For example, this command creates a container and limits the write rate to `1mb` per second for `/dev/sda` :

```
$ docker run -it --device-write-bps /dev/sda:1mb ubuntu
```

Both flags take limits in the `<device-path>:<limit>[unit]` format. Both read and write rates must be a positive integer. You can specify the rate in `kb` (kilobytes), `mb` (megabytes), or `gb` (gigabytes).

The `--device-read-iops` flag limits read rate (IO per second) from a device. For example, this command creates a container and limits the read rate to `1000` IO per second from `/dev/sda` :

```
$ docker run -ti --device-read-iops /dev/sda:1000 ubuntu
```

The `--device-write-iops` flag limits write rate (IO per second) to a device. For example, this command creates a container and limits the write rate to `1000` IO per second to `/dev/sda` :

```
$ docker run -ti --device-write-iops /dev/sda:1000 ubuntu
```

Both flags take limits in the `<device-path>:<limit>` format. Both read and write rates must be a positive integer.

Additional groups

```
--group-add: Add additional groups to run as
```

By default, the docker container process runs with the supplementary groups looked up for the specified user. If one wants to add more to that list of groups, then one can use this flag:

```
$ docker run --rm --group-add audio --group-add nogroup --group-add 777 busybox id
uid=0(root) gid=0(root) groups=10(wheel),29(audio),99(nogroup),777
```

Runtime privilege and Linux capabilities

```
--cap-add: Add Linux capabilities
--cap-drop: Drop Linux capabilities
--privileged=false: Give extended privileges to this container
--device=[]: Allows you to run devices inside the container without the --privileged flag.
```

By default, Docker containers are "unprivileged" and cannot, for example, run a Docker daemon inside a Docker container. This is because by default a container is not allowed to access any devices, but a "privileged" container is given access to all devices (see the documentation on cgroups devices (https://www.kernel.org/doc/Documentation/cgroup-v1/devices.txt)).

When the operator executes `docker run --privileged`, Docker will enable access to all devices on the host as well as set some configuration in AppArmor or SELinux to allow the container nearly all the same access to the host as processes running outside containers on the host. Additional information about running with `--privileged` is available on the Docker Blog (http://blog.docker.com/2013/09/docker-can-now-run-within-docker/).

If you want to limit access to a specific device or devices you can use the `--device` flag. It allows you to specify one or more devices that will be accessible within the container.

```
$ docker run --device=/dev/snd:/dev/snd ...
```

By default, the container will be able to `read`, `write`, and `mknod` these devices. This can be overridden using a third `:rwm` set of options to each `--device` flag:

```
$ docker run --device=/dev/sda:/dev/xvdc --rm -it ubuntu fdisk  /dev/xvdc

Command (m for help): q
$ docker run --device=/dev/sda:/dev/xvdc:r --rm -it ubuntu fdisk  /dev/xvdc
You will not be able to write the partition table.

Command (m for help): q

$ docker run --device=/dev/sda:/dev/xvdc:w --rm -it ubuntu fdisk  /dev/xvdc
    crash....

$ docker run --device=/dev/sda:/dev/xvdc:m --rm -it ubuntu fdisk  /dev/xvdc
fdisk: unable to open /dev/xvdc: Operation not permitted
```

In addition to `--privileged` , the operator can have fine grain control over the capabilities using `--cap-add` and `--cap-drop` . By default, Docker has a default list of capabilities that are kept. The following table lists the Linux capability options which are allowed by default and can be dropped.

Capability Key	Capability Description
SETPCAP	Modify process capabilities.
MKNOD	Create special files using mknod(2).
AUDIT_WRITE	Write records to kernel auditing log.
CHOWN	Make arbitrary changes to file UIDs and GIDs (see chown(2)).
NET_RAW	Use RAW and PACKET sockets.
DAC_OVERRIDE	Bypass file read, write, and execute permission checks.
FOWNER	Bypass permission checks on operations that normally require the file system UID of the process to match the UID of the file.
FSETID	Don’t clear set-user-ID and set-group-ID permission bits when a file is modified.
KILL	Bypass permission checks for sending signals.
SETGID	Make arbitrary manipulations of process GIDs and supplementary GID list.
SETUID	Make arbitrary manipulations of process UIDs.
NET_BIND_SERVICE	Bind a socket to internet domain privileged ports (port numbers less than 1024).
SYS_CHROOT	Use chroot(2), change root directory.
SETFCAP	Set file capabilities.

The next table shows the capabilities which are not granted by default and may be added.

Capability Key	Capability Description

Capability Key	Capability Description
SYS_MODULE	Load and unload kernel modules.
SYS_RAWIO	Perform I/O port operations (iopl(2) and ioperm(2)).
SYS_PACCT	Use acct(2), switch process accounting on or off.
SYS_ADMIN	Perform a range of system administration operations.
SYS_NICE	Raise process nice value (nice(2), setpriority(2)) and change the nice value for arbitrary processes.
SYS_RESOURCE	Override resource Limits.
SYS_TIME	Set system clock (settimeofday(2), stime(2), adjtimex(2)); set real-time (hardware) clock.
SYS_TTY_CONFIG	Use vhangup(2); employ various privileged ioctl(2) operations on virtual terminals.
AUDIT_CONTROL	Enable and disable kernel auditing; change auditing filter rules; retrieve auditing status and filtering rules.
MAC_ADMIN	Allow MAC configuration or state changes. Implemented for the Smack LSM.
MAC_OVERRIDE	Override Mandatory Access Control (MAC). Implemented for the Smack Linux Security Module (LSM).
NET_ADMIN	Perform various network-related operations.
SYSLOG	Perform privileged syslog(2) operations.
DAC_READ_SEARCH	Bypass file read permission checks and directory read and execute permission checks.
LINUX_IMMUTABLE	Set the FS_APPEND_FL and FS_IMMUTABLE_FL i-node flags.
NET_BROADCAST	Make socket broadcasts, and listen to multicasts.
IPC_LOCK	Lock memory (mlock(2), mlockall(2), mmap(2), shmctl(2)).
IPC_OWNER	Bypass permission checks for operations on System V IPC objects.
SYS_PTRACE	Trace arbitrary processes using ptrace(2).
SYS_BOOT	Use reboot(2) and kexec_load(2), reboot and load a new kernel for later execution.
LEASE	Establish leases on arbitrary files (see fcntl(2)).
WAKE_ALARM	Trigger something that will wake up the system.
BLOCK_SUSPEND	Employ features that can block system suspend.

Further reference information is available on the capabilities(7) - Linux man page (http://man7.org/linux/man-pages/man7/capabilities.7.html)

Both flags support the value `ALL`, so if the operator wants to have all capabilities but `MKNOD` they could use:

```
$ docker run --cap-add=ALL --cap-drop=MKNOD ...
```

For interacting with the network stack, instead of using `--privileged` they should use `--cap-add=NET_ADMIN` to modify the network interfaces.

```
$ docker run -it --rm  ubuntu:14.04 ip link add dummy0 type dummy
RTNETLINK answers: Operation not permitted
$ docker run -it --rm --cap-add=NET_ADMIN ubuntu:14.04 ip link add dummy0 type dummy
```

To mount a FUSE based filesystem, you need to combine both `--cap-add` and `--device`:

```
$ docker run --rm -it --cap-add SYS_ADMIN sshfs sshfs sven@10.10.10.20:/home/sven /mnt
fuse: failed to open /dev/fuse: Operation not permitted
$ docker run --rm -it --device /dev/fuse sshfs sshfs sven@10.10.10.20:/home/sven /mnt
fusermount: mount failed: Operation not permitted
$ docker run --rm -it --cap-add SYS_ADMIN --device /dev/fuse sshfs
# sshfs sven@10.10.10.20:/home/sven /mnt
The authenticity of host '10.10.10.20 (10.10.10.20)' can't be established.
ECDSA key fingerprint is 25:34:85:75:25:b0:17:46:05:19:04:93:b5:dd:5f:c6.
Are you sure you want to continue connecting (yes/no)? yes
sven@10.10.10.20's password:
root@30aa0cfaf1b5:/# ls -la /mnt/src/docker
total 1516
drwxrwxr-x 1 1000 1000   4096 Dec  4 06:08 .
drwxrwxr-x 1 1000 1000   4096 Dec  4 11:46 ..
-rw-rw-r-- 1 1000 1000     16 Oct  8 00:09 .dockerignore
-rwxrwxr-x 1 1000 1000    464 Oct  8 00:09 .drone.yml
drwxrwxr-x 1 1000 1000   4096 Dec  4 06:11 .git
-rw-rw-r-- 1 1000 1000    461 Dec  4 06:08 .gitignore
....
```

The default seccomp profile will adjust to the selected capabilities, in order to allow use of facilities allowed by the capabilities, so you should not have to adjust this, since Docker 1.12. In Docker 1.10 and 1.11 this did not happen and it may be necessary to use a custom seccomp profile or use `--security-opt seccomp=unconfined` when adding capabilities.

Logging drivers (--log-driver)

The container can have a different logging driver than the Docker daemon. Use the `--log-driver=VALUE` with the `docker run` command to configure the container's logging driver. The following options are supported:

Driver	Description
`none`	Disables any logging for the container. `docker logs` won't be available with this driver.

Driver	Description
`json-file`	Default logging driver for Docker. Writes JSON messages to file. No logging options are supported for this driver.
`syslog`	Syslog logging driver for Docker. Writes log messages to syslog.
`journald`	Journald logging driver for Docker. Writes log messages to `journald` .
`gelf`	Graylog Extended Log Format (GELF) logging driver for Docker. Writes log messages to a GELF endpoint likeGraylog or Logstash.
`fluentd`	Fluentd logging driver for Docker. Writes log messages to `fluentd` (forward input).
`awslogs`	Amazon CloudWatch Logs logging driver for Docker. Writes log messages to Amazon CloudWatch Logs
`splunk`	Splunk logging driver for Docker. Writes log messages to `splunk` using Event Http Collector.

The `docker logs` command is available only for the `json-file` and `journald` logging drivers. For detailed information on working with logging drivers, see Configure logging drivers (https://docs.docker.com/config/containers/logging/configure/).

Overriding Dockerfile image defaults

When a developer builds an image from a *Dockerfile* (https://docs.docker.com/engine/reference/builder/) or when she commits it, the developer can set a number of default parameters that take effect when the image starts up as a container.

Four of the Dockerfile commands cannot be overridden at runtime: `FROM` , `MAINTAINER` , `RUN` , and `ADD` . Everything else has a corresponding override in `docker run` . We'll go through what the developer might have set in each Dockerfile instruction and how the operator can override that setting.

- CMD (Default Command or Options) (/engine/reference/run/#cmd-default-command-or-options)
- ENTRYPOINT (Default Command to Execute at Runtime) (/engine/reference/run/#entrypoint-default-command-to-execute-at-runtime)
- EXPOSE (Incoming Ports) (/engine/reference/run/#expose-incoming-ports)
- ENV (Environment Variables) (/engine/reference/run/#env-environment-variables)
- HEALTHCHECK (/engine/reference/run/#healthcheck)
- VOLUME (Shared Filesystems) (/engine/reference/run/#volume-shared-filesystems)
- USER (/engine/reference/run/#user)
- WORKDIR (/engine/reference/run/#workdir)

CMD (default command or options)

Recall the optional `COMMAND` in the Docker commandline:

```
$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

This command is optional because the person who created the `IMAGE` may have already provided a default `COMMAND` using the Dockerfile `CMD` instruction. As the operator (the person running a container from the image), you can override that `CMD` instruction just by specifying a new `COMMAND`.

If the image also specifies an `ENTRYPOINT` then the `CMD` or `COMMAND` get appended as arguments to the `ENTRYPOINT`.

ENTRYPOINT (default command to execute at runtime)

```
--entrypoint="": Overwrite the default entrypoint set by the image
```

The `ENTRYPOINT` of an image is similar to a `COMMAND` because it specifies what executable to run when the container starts, but it is (purposely) more difficult to override. The `ENTRYPOINT` gives a container its default nature or behavior, so that when you set an `ENTRYPOINT` you can run the container *as if it were that binary*, complete with default options, and you can pass in more options via the `COMMAND`. But, sometimes an operator may want to run something else inside the container, so you can override the default `ENTRYPOINT` at runtime by using a string to specify the new `ENTRYPOINT`. Here is an example of how to run a shell in a container that has been set up to automatically run something else (like `/usr/bin/redis-server`):

```
$ docker run -it --entrypoint /bin/bash example/redis
```

or two examples of how to pass more parameters to that ENTRYPOINT:

```
$ docker run -it --entrypoint /bin/bash example/redis -c ls -l
$ docker run -it --entrypoint /usr/bin/redis-cli example/redis --help
```

You can reset a containers entrypoint by passing an empty string, for example:

```
$ docker run -it --entrypoint="" mysql bash
```

> **Note**: Passing `--entrypoint` will clear out any default command set on the image (i.e. any `CMD` instruction in the Dockerfile used to build it).

EXPOSE (incoming ports)

The following `run` command options work with container networking:

```
--expose=[]: Expose a port or a range of ports inside the container.
             These are additional to those exposed by the `EXPOSE` instruction
-P         : Publish all exposed ports to the host interfaces
-p=[]      : Publish a container's port or a range of ports to the host
               format: ip:hostPort:containerPort | ip::containerPort | hostPort:containerPort | co
               Both hostPort and containerPort can be specified as a
               range of ports. When specifying ranges for both, the
               number of container ports in the range must match the
               number of host ports in the range, for example:
                   -p 1234-1236:1234-1236/tcp

               When specifying a range for hostPort only, the
               containerPort must not be a range.  In this case the
               container port is published somewhere within the
               specified hostPort range. (e.g., `-p 1234-1236:1234/tcp`)

               (use 'docker port' to see the actual mapping)

--link=""  : Add link to another container (<name or id>:alias or <name or id>)
```

With the exception of the `EXPOSE` directive, an image developer hasn't got much control over networking. The `EXPOSE` instruction defines the initial incoming ports that provide services. These ports are available to processes inside the container. An operator can use the `--expose` option to add to the exposed ports.

To expose a container's internal port, an operator can start the container with the `-P` or `-p` flag. The exposed port is accessible on the host and the ports are available to any client that can reach the host.

The `-P` option publishes all the ports to the host interfaces. Docker binds each exposed port to a random port on the host. The range of ports are within an *ephemeral port range* defined by `/proc/sys/net/ipv4/ip_local_port_range` . Use the `-p` flag to explicitly map a single port or range of ports.

The port number inside the container (where the service listens) does not need to match the port number exposed on the outside of the container (where clients connect). For example, inside the container an HTTP service is listening on port 80 (and so the image developer specifies `EXPOSE 80` in the Dockerfile). At runtime, the port might be bound to 42800 on the host. To find the mapping between the host ports and the exposed ports, use `docker port` .

If the operator uses `--link` when starting a new client container in the default bridge network, then the client container can access the exposed port via a private networking interface. If `--link` is used when starting a container in a user-defined network as described in *Networking overview* (https://docs.docker.com/network/), it will provide a named alias for the container being linked to.

ENV (environment variables)

Docker automatically sets some environment variables when creating a Linux container. Docker does not set any environment variables when creating a Windows container.

The following environment variables are set for Linux containers:

Variable	Value
`HOME`	Set based on the value of `USER`

Variable	Value
`HOSTNAME`	The hostname associated with the container
`PATH`	Includes popular directories, such as `/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin`
`TERM`	`xterm` if the container is allocated a pseudo-TTY

Additionally, the operator can **set any environment variable** in the container by using one or more `-e` flags, even overriding those mentioned above, or already defined by the developer with a Dockerfile `ENV` . If the operator names an environment variable without specifying a value, then the current value of the named variable is propagated into the container's environment:

```
$ export today=Wednesday
$ docker run -e "deep=purple" -e today --rm alpine env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=d2219b854598
deep=purple
today=Wednesday
HOME=/root
```

```
PS C:\> docker run --rm -e "foo=bar" microsoft/nanoserver cmd /s /c set
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\ContainerAdministrator\AppData\Roaming
CommonProgramFiles=C:\Program Files\Common Files
CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
CommonProgramW6432=C:\Program Files\Common Files
COMPUTERNAME=C2FAEFCC8253
ComSpec=C:\Windows\system32\cmd.exe
foo=bar
LOCALAPPDATA=C:\Users\ContainerAdministrator\AppData\Local
NUMBER_OF_PROCESSORS=8
OS=Windows_NT
Path=C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell
PATHEXT=.COM;.EXE;.BAT;.CMD
PROCESSOR_ARCHITECTURE=AMD64
PROCESSOR_IDENTIFIER=Intel64 Family 6 Model 62 Stepping 4, GenuineIntel
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=3e04
ProgramData=C:\ProgramData
ProgramFiles=C:\Program Files
ProgramFiles(x86)=C:\Program Files (x86)
ProgramW6432=C:\Program Files
PROMPT=$P$G
PUBLIC=C:\Users\Public
SystemDrive=C:
SystemRoot=C:\Windows
TEMP=C:\Users\ContainerAdministrator\AppData\Local\Temp
TMP=C:\Users\ContainerAdministrator\AppData\Local\Temp
USERDOMAIN=User Manager
USERNAME=ContainerAdministrator
USERPROFILE=C:\Users\ContainerAdministrator
windir=C:\Windows
```

Similarly the operator can set the **HOSTNAME** (Linux) or **COMPUTERNAME** (Windows) with `-h` .

HEALTHCHECK

```
  --health-cmd            Command to run to check health
  --health-interval       Time between running the check
  --health-retries        Consecutive failures needed to report unhealthy
  --health-timeout        Maximum time to allow one check to run
  --health-start-period   Start period for the container to initialize before starting health-retr
  --no-healthcheck        Disable any container-specified HEALTHCHECK
```

Example:

```
$ docker run --name=test -d \
    --health-cmd='stat /etc/passwd || exit 1' \
    --health-interval=2s \
    busybox sleep 1d
$ sleep 2; docker inspect --format='{{.State.Health.Status}}' test
healthy
$ docker exec test rm /etc/passwd
$ sleep 2; docker inspect --format='{{json .State.Health}}' test
{
  "Status": "unhealthy",
  "FailingStreak": 3,
  "Log": [
    {
      "Start": "2016-05-25T17:22:04.635478668Z",
      "End": "2016-05-25T17:22:04.7272552Z",
      "ExitCode": 0,
      "Output": "  File: /etc/passwd\n  Size: 334       \tBlocks: 8          IO Block: 4096   regu
    },
    {
      "Start": "2016-05-25T17:22:06.732900633Z",
      "End": "2016-05-25T17:22:06.822168935Z",
      "ExitCode": 0,
      "Output": "  File: /etc/passwd\n  Size: 334       \tBlocks: 8          IO Block: 4096   regu
    },
    {
      "Start": "2016-05-25T17:22:08.823956535Z",
      "End": "2016-05-25T17:22:08.897359124Z",
      "ExitCode": 1,
      "Output": "stat: can't stat '/etc/passwd': No such file or directory\n"
    },
    {
      "Start": "2016-05-25T17:22:10.898802931Z",
      "End": "2016-05-25T17:22:10.969631866Z",
      "ExitCode": 1,
      "Output": "stat: can't stat '/etc/passwd': No such file or directory\n"
    },
    {
      "Start": "2016-05-25T17:22:12.971033523Z",
      "End": "2016-05-25T17:22:13.082015516Z",
      "ExitCode": 1,
      "Output": "stat: can't stat '/etc/passwd': No such file or directory\n"
    }
  ]
}
```

The health status is also displayed in the `docker ps` output.

TMPFS (mount tmpfs filesystems)

```
--tmpfs=[]: Create a tmpfs mount with: container-dir[:<options>],
            where the options are identical to the Linux
            'mount -t tmpfs -o' command.
```

The example below mounts an empty tmpfs into the container with the `rw`, `noexec`, `nosuid`, and `size=65536k` options.

```
$ docker run -d --tmpfs /run:rw,noexec,nosuid,size=65536k my_image
```

VOLUME (shared filesystems)

```
-v, --volume=[host-src:]container-dest[:<options>]: Bind mount a volume.
The comma-delimited `options` are [rw|ro], [z|Z],
[[r]shared|[r]slave|[r]private], and [nocopy].
The 'host-src' is an absolute path or a name value.

If neither 'rw' or 'ro' is specified then the volume is mounted in
read-write mode.

The `nocopy` mode is used to disable automatically copying the requested volume
path in the container to the volume storage location.
For named volumes, `copy` is the default mode. Copy modes are not supported
for bind-mounted volumes.

--volumes-from="": Mount all volumes from the given container(s)
```

> **Note**: When using systemd to manage the Docker daemon's start and stop, in the systemd unit file there is an option to control mount propagation for the Docker daemon itself, called `MountFlags`. The value of this setting may cause Docker to not see mount propagation changes made on the mount point. For example, if this value is `slave`, you may not be able to use the `shared` or `rshared` propagation on a volume.

The volumes commands are complex enough to have their own documentation in section *Use volumes* (https://docs.docker.com/storage/volumes/). A developer can define one or more `VOLUME`'s associated with an image, but only the operator can give access from one container to another (or from a container to a volume mounted on the host).

The `container-dest` must always be an absolute path such as `/src/docs`. The `host-src` can either be an absolute path or a `name` value. If you supply an absolute path for the `host-dir`, Docker bind-mounts to the path you specify. If you supply a `name`, Docker creates a named volume by that `name`.

A `name` value must start with an alphanumeric character, followed by `a-z0-9`, `_` (underscore), `.` (period) or `-` (hyphen). An absolute path starts with a `/` (forward slash).

For example, you can specify either `/foo` or `foo` for a `host-src` value. If you supply the `/foo` value, Docker creates a bind mount. If you supply the `foo` specification, Docker creates a named volume.

USER

`root` (id = 0) is the default user within a container. The image developer can create additional users. Those users are accessible by name. When passing a numeric ID, the user does not have to exist in the container.

The developer can set a default user to run the first process with the Dockerfile `USER` instruction. When starting a container, the operator can override the `USER` instruction by passing the `-u` option.

```
-u="", --user="": Sets the username or UID used and optionally the groupname or GID for the specif

The followings examples are all valid:
--user=[ user | user:group | uid | uid:gid | user:gid | uid:group ]
```

> **Note:** if you pass a numeric uid, it must be in the range of 0-2147483647.

WORKDIR

The default working directory for running binaries within a container is the root directory (`/`), but the developer can set a different default with the Dockerfile `WORKDIR` command. The operator can override this with:

```
-w="": Working directory inside the container
```

docker (https://docs.docker.com/glossary/?term=docker), run (https://docs.docker.com/glossary/?term=run), configure (https://docs.docker.com/glossary/?term=configure), runtime (https://docs.docker.com/glossary/?term=runtime)