docker build

Estimated reading time: 22 minutes

Description

Build an image from a Dockerfile

Usage

```
docker build [OPTIONS] PATH | URL | -
```

Options

Name, shorthand	Default	Description
`--add-host`		Add a custom host-to-IP mapping (host:ip)
`--build-arg`		Set build-time variables
`--cache-from`		Images to consider as cache sources
`--cgroup-parent`		Optional parent cgroup for the container
`--compress`		Compress the build context using gzip
`--cpu-period`		Limit the CPU CFS (Completely Fair Scheduler) period
`--cpu-quota`		Limit the CPU CFS (Completely Fair Scheduler) quota
`--cpu-shares , -c`		CPU shares (relative weight)
`--cpuset-cpus`		CPUs in which to allow execution (0-3, 0,1)
`--cpuset-mems`		MEMs in which to allow execution (0-3, 0,1)
`--disable-content-trust`	`true`	Skip image verification
`--file , -f`		Name of the Dockerfile (Default is 'PATH/Dockerfile')
`--force-rm`		Always remove intermediate containers
`--iidfile`		Write the image ID to the file
`--isolation`		Container isolation technology
`--label`		Set metadata for an image
`--memory , -m`		Memory limit
`--memory-swap`		Swap limit equal to memory plus swap: '-1' to enable unlimited swap
`--network`		**API 1.25+** (https://docs.docker.com/engine/api/v1.25/) Set the networking mode for the RUN instructions during build

Name, shorthand	Default	Description
`--no-cache`		Do not use cache when building the image
`--platform`		**experimental (daemon)** (https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file) **API 1.32+** (https://docs.docker.com/engine/api/v1.32/) Set platform if server is multi-platform capable
`--progress`	`auto`	Set type of progress output (auto, plain, tty). Use plain to show container output
`--pull`		Always attempt to pull a newer version of the image
`--quiet , -q`		Suppress the build output and print image ID on success
`--rm`	`true`	Remove intermediate containers after a successful build
`--secret`		**API 1.39+** (https://docs.docker.com/engine/api/v1.39/) Secret file to expose to the build (only if BuildKit enabled): id=mysecret,src=/local/secret
`--security-opt`		Security options
`--shm-size`		Size of /dev/shm
`--squash`		**experimental (daemon)** (https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file) **API 1.25+** (https://docs.docker.com/engine/api/v1.25/) Squash newly built layers into a single new layer
`--ssh`		**API 1.39+** (https://docs.docker.com/engine/api/v1.39/) SSH agent socket or keys to expose to the build (only if BuildKit enabled) (format: default|[=|[,]])
`--stream`		**experimental (daemon)** (https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file) **API 1.31+** (https://docs.docker.com/engine/api/v1.31/) Stream attaches to server to negotiate build context
`--tag , -t`		Name and optionally a tag in the 'name:tag' format
`--target`		Set the target build stage to build.
`--ulimit`		Ulimit options

Parent command

Command	Description
docker (https://docs.docker.com/engine/reference/commandline/docker)	The base command for the Docker CLI.

Extended description

The `docker build` command builds Docker images from a Dockerfile and a "context". A build's context is the set of files located in the specified `PATH` or `URL`. The build process can refer to any of the files in the context. For example, your build can use a *COPY* (https://docs.docker.com/engine/reference/builder/#copy) instruction to reference a file in the context.

The `URL` parameter can refer to three kinds of resources: Git repositories, pre-packaged tarball contexts and

plain text files.

Git repositories

When the `URL` parameter points to the location of a Git repository, the repository acts as the build context. The system recursively fetches the repository and its submodules. The commit history is not preserved. A repository is first pulled into a temporary directory on your local host. After that succeeds, the directory is sent to the Docker daemon as the context. Local copy gives you the ability to access private repositories using local user credentials, VPN's, and so forth.

> **Note:** If the `URL` parameter contains a fragment the system will recursively clone the repository and its submodules using a `git clone --recursive` command.

Git URLs accept context configuration in their fragment section, separated by a colon `:`. The first part represents the reference that Git will check out, and can be either a branch, a tag, or a remote reference. The second part represents a subdirectory inside the repository that will be used as a build context.

For example, run this command to use a directory called `docker` in the branch `container`:

```
$ docker build https://github.com/docker/rootfs.git#container:docker
```

The following table represents all the valid suffixes with their build contexts:

Build Syntax Suffix	Commit Used	Build Context Used
`myrepo.git`	`refs/heads/master`	`/`
`myrepo.git#mytag`	`refs/tags/mytag`	`/`
`myrepo.git#mybranch`	`refs/heads/mybranch`	`/`
`myrepo.git#pull/42/head`	`refs/pull/42/head`	`/`
`myrepo.git#:myfolder`	`refs/heads/master`	`/myfolder`
`myrepo.git#master:myfolder`	`refs/heads/master`	`/myfolder`
`myrepo.git#mytag:myfolder`	`refs/tags/mytag`	`/myfolder`
`myrepo.git#mybranch:myfolder`	`refs/heads/mybranch`	`/myfolder`

Tarball contexts

If you pass an URL to a remote tarball, the URL itself is sent to the daemon:

```
$ docker build http://server/context.tar.gz
```

The download operation will be performed on the host the Docker daemon is running on, which is not necessarily the same host from which the build command is being issued. The Docker daemon will fetch `context.tar.gz` and use it as the build context. Tarball contexts must be tar archives conforming to the standard `tar` UNIX format and can be compressed with any one of the 'xz', 'bzip2', 'gzip' or 'identity' (no compression) formats.

Text files

Instead of specifying a context, you can pass a single `Dockerfile` in the `URL` or pipe the file in via `STDIN`. To pipe a `Dockerfile` from `STDIN`:

```
$ docker build - < Dockerfile
```

With Powershell on Windows, you can run:

```
Get-Content Dockerfile | docker build -
```

If you use `STDIN` or specify a `URL` pointing to a plain text file, the system places the contents into a file called `Dockerfile`, and any `-f`, `--file` option is ignored. In this scenario, there is no context.

By default the `docker build` command will look for a `Dockerfile` at the root of the build context. The `-f`, `--file`, option lets you specify the path to an alternative file to use instead. This is useful in cases where the same set of files are used for multiple builds. The path must be to a file within the build context. If a relative path is specified then it is interpreted as relative to the root of the context.

In most cases, it's best to put each Dockerfile in an empty directory. Then, add to that directory only the files needed for building the Dockerfile. To increase the build's performance, you can exclude files and directories by adding a `.dockerignore` file to that directory as well. For information on creating one, see the .dockerignore file (https://docs.docker.com/engine/reference/builder/#dockerignore-file).

If the Docker client loses connection to the daemon, the build is canceled. This happens if you interrupt the Docker client with `CTRL-c` or if the Docker client is killed for any reason. If the build initiated a pull which is still running at the time the build is cancelled, the pull is cancelled as well.

Examples

Build with PATH

```
$ docker build .

Uploading context 10240 bytes
Step 1/3 : FROM busybox
Pulling repository busybox
 ---> e9aa60c60128MB/2.284 MB (100%) endpoint: https://cdn-registry-1.docker.io/v1/
Step 2/3 : RUN ls -lh /
 ---> Running in 9c9e81692ae9
total 24
drwxr-xr-x    2 root     root        4.0K Mar 12  2013 bin
drwxr-xr-x    5 root     root        4.0K Oct 19 00:19 dev
drwxr-xr-x    2 root     root        4.0K Oct 19 00:19 etc
drwxr-xr-x    2 root     root        4.0K Nov 15 23:34 lib
lrwxrwxrwx    1 root     root           3 Mar 12  2013 lib64 -> lib
dr-xr-xr-x  116 root     root           0 Nov 15 23:34 proc
lrwxrwxrwx    1 root     root           3 Mar 12  2013 sbin -> bin
dr-xr-xr-x   13 root     root           0 Nov 15 23:34 sys
drwxr-xr-x    2 root     root        4.0K Mar 12  2013 tmp
drwxr-xr-x    2 root     root        4.0K Nov 15 23:34 usr
 ---> b35f4035db3f
Step 3/3 : CMD echo Hello world
 ---> Running in 02071fceb21b
 ---> f52f38b7823e
Successfully built f52f38b7823e
Removing intermediate container 9c9e81692ae9
Removing intermediate container 02071fceb21b
```

This example specifies that the `PATH` is `.`, and so all the files in the local directory get `tar` d and sent to the Docker daemon. The `PATH` specifies where to find the files for the "context" of the build on the Docker daemon. Remember that the daemon could be running on a remote machine and that no parsing of the

Dockerfile happens at the client side (where you're running `docker build`). That means that *all* the files at `PATH` get sent, not just the ones listed to *ADD* (https://docs.docker.com/engine/reference/builder/#add) in the Dockerfile.

The transfer of context from the local machine to the Docker daemon is what the `docker` client means when you see the "Sending build context" message.

If you wish to keep the intermediate containers after the build is complete, you must use `--rm=false`. This does not affect the build cache.

Build with URL

```
$ docker build github.com/creack/docker-firefox
```

This will clone the GitHub repository and use the cloned repository as context. The Dockerfile at the root of the repository is used as Dockerfile. You can specify an arbitrary Git repository by using the `git://` or `git@` scheme.

```
$ docker build -f ctx/Dockerfile http://server/ctx.tar.gz

Downloading context: http://server/ctx.tar.gz [==================>]    240 B/240 B
Step 1/3 : FROM busybox
 ---> 8c2e06607696
Step 2/3 : ADD ctx/container.cfg /
 ---> e7829950cee3
Removing intermediate container b35224abf821
Step 3/3 : CMD /bin/ls
 ---> Running in fbc63d321d73
 ---> 3286931702ad
Removing intermediate container fbc63d321d73
Successfully built 377c409b35e4
```

This sends the URL `http://server/ctx.tar.gz` to the Docker daemon, which downloads and extracts the referenced tarball. The `-f ctx/Dockerfile` parameter specifies a path inside `ctx.tar.gz` to the `Dockerfile` that is used to build the image. Any `ADD` commands in that `Dockerfile` that refers to local paths must be relative to the root of the contents inside `ctx.tar.gz`. In the example above, the tarball contains a directory `ctx/`, so the `ADD ctx/container.cfg /` operation works as expected.

Build with -

```
$ docker build - < Dockerfile
```

This will read a Dockerfile from `STDIN` without context. Due to the lack of a context, no contents of any local directory will be sent to the Docker daemon. Since there is no context, a Dockerfile `ADD` only works if it refers to a remote URL.

```
$ docker build - < context.tar.gz
```

This will build an image for a compressed context read from `STDIN`. Supported formats are: bzip2, gzip and xz.

Use a .dockerignore file

```
$ docker build .

Uploading context 18.829 MB
Uploading context
Step 1/2 : FROM busybox
 ---> 769b9341d937
Step 2/2 : CMD echo Hello world
 ---> Using cache
 ---> 99cc1ad10469
Successfully built 99cc1ad10469
$ echo ".git" > .dockerignore
$ docker build .
Uploading context  6.76 MB
Uploading context
Step 1/2 : FROM busybox
 ---> 769b9341d937
Step 2/2 : CMD echo Hello world
 ---> Using cache
 ---> 99cc1ad10469
Successfully built 99cc1ad10469
```

This example shows the use of the `.dockerignore` file to exclude the `.git` directory from the context. Its effect can be seen in the changed size of the uploaded context. The builder reference contains detailed information on creating a .dockerignore file (https://docs.docker.com/engine/reference/builder/#dockerignore-file)

Tag an image (-t)

```
$ docker build -t vieux/apache:2.0 .
```

This will build like the previous example, but it will then tag the resulting image. The repository name will be `vieux/apache` and the tag will be `2.0` . Read more about valid tags (https://docs.docker.com/engine/reference/commandline/tag/).

You can apply multiple tags to an image. For example, you can apply the `latest` tag to a newly built image and add another tag that references a specific version. For example, to tag an image both as `whenry/fedora-jboss:latest` and `whenry/fedora-jboss:v2.1` , use the following:

```
$ docker build -t whenry/fedora-jboss:latest -t whenry/fedora-jboss:v2.1 .
```

Specify a Dockerfile (-f)

```
$ docker build -f Dockerfile.debug .
```

This will use a file called `Dockerfile.debug` for the build instructions instead of `Dockerfile` .

```
$ curl example.com/remote/Dockerfile | docker build -f - .
```

The above command will use the current directory as the build context and read a Dockerfile from stdin.

```
$ docker build -f dockerfiles/Dockerfile.debug -t myapp_debug .
$ docker build -f dockerfiles/Dockerfile.prod  -t myapp_prod .
```

The above commands will build the current build context (as specified by the `.`) twice, once using a debug version of a `Dockerfile` and once using a production version.

```
$ cd /home/me/myapp/some/dir/really/deep
$ docker build -f /home/me/myapp/dockerfiles/debug /home/me/myapp
$ docker build -f ../../../../dockerfiles/debug /home/me/myapp
```

These two `docker build` commands do the exact same thing. They both use the contents of the `debug` file instead of looking for a `Dockerfile` and will use `/home/me/myapp` as the root of the build context. Note that `debug` is in the directory structure of the build context, regardless of how you refer to it on the command line.

> **Note:** `docker build` will return a `no such file or directory` error if the file or directory does not exist in the uploaded context. This may happen if there is no context, or if you specify a file that is elsewhere on the Host system. The context is limited to the current directory (and its children) for security reasons, and to ensure repeatable builds on remote Docker hosts. This is also the reason why `ADD ../file` will not work.

Use a custom parent cgroup (--cgroup-parent)

When `docker build` is run with the `--cgroup-parent` option the containers used in the build will be run with the corresponding `docker run` flag (https://docs.docker.com/engine/reference/run/#specifying-custom-cgroups).

Set ulimits in container (--ulimit)

Using the `--ulimit` option with `docker build` will cause each build step's container to be started using those `--ulimit` flag values (https://docs.docker.com/engine/reference/commandline/run/#set-ulimits-in-container-ulimit).

Set build-time variables (--build-arg)

You can use `ENV` instructions in a Dockerfile to define variable values. These values persist in the built image. However, often persistence is not what you want. Users want to specify variables differently depending on which host they build an image on.

A good example is `http_proxy` or source versions for pulling intermediate files. The `ARG` instruction lets Dockerfile authors define values that users can set at build-time using the `--build-arg` flag:

```
$ docker build --build-arg HTTP_PROXY=http://10.20.30.2:1234 --build-arg FTP_PROXY=http://40.50
.60.5:4567 .
```

This flag allows you to pass the build-time variables that are accessed like regular environment variables in the `RUN` instruction of the Dockerfile. Also, these values don't persist in the intermediate or final images like `ENV` values do. You must add `--build-arg` for each build argument.

Using this flag will not alter the output you see when the `ARG` lines from the Dockerfile are echoed during the build process.

For detailed information on using `ARG` and `ENV` instructions, see the Dockerfile reference (https://docs.docker.com/engine/reference/builder/).

You may also use the `--build-arg` flag without a value, in which case the value from the local environment will be propagated into the Docker container being built:

```
$ export HTTP_PROXY=http://10.20.30.2:1234
$ docker build --build-arg HTTP_PROXY .
```

This is similar to how `docker run -e` works. Refer to the `docker run` documentation (https://docs.docker.com/engine/reference/commandline/run/#set-environment-variables--e---env---env-file) for more information.

Optional security options (--security-opt)

This flag is only supported on a daemon running on Windows, and only supports the `credentialspec` option. The `credentialspec` must be in the format `file://spec.txt` or `registry://keyname` .

Specify isolation technology for container (--isolation)

This option is useful in situations where you are running Docker containers on Windows. The `--isolation=<value>` option sets a container's isolation technology. On Linux, the only supported is the `default` option which uses Linux namespaces. On Microsoft Windows, you can specify these values:

Value	Description
`default`	Use the value specified by the Docker daemon's `--exec-opt` . If the `daemon` does not specify an isolation technology, Microsoft Windows uses `process` as its default value.
`process`	Namespace isolation only.
`hyperv`	Hyper-V hypervisor partition-based isolation.

Specifying the `--isolation` flag without a value is the same as setting `--isolation="default"` .

Add entries to container hosts file (--add-host)

You can add other hosts into a container's `/etc/hosts` file by using one or more `--add-host` flags. This example adds a static address for a host named `docker` :

```
$ docker build --add-host=docker:10.180.0.1 .
```

Specifying target build stage (--target)

When building a Dockerfile with multiple build stages, `--target` can be used to specify an intermediate build stage by name as a final stage for the resulting image. Commands after the target stage will be skipped.

```
FROM debian AS build-env
...

FROM alpine AS production-env
...
```

```
$ docker build -t mybuildimage --target build-env .
```

Squash an image's layers (--squash) (experimental)

OVERVIEW

Once the image is built, squash the new layers into a new image with a single new layer. Squashing does not destroy any existing image, rather it creates a new image with the content of the squashed layers. This effectively makes it look like all `Dockerfile` commands were created with a single layer. The build cache is preserved with this method.

The `--squash` option is an experimental feature, and should not be considered stable.

Squashing layers can be beneficial if your Dockerfile produces multiple layers modifying the same files, for example, file that are created in one step, and removed in another step. For other use-cases, squashing images may actually have a negative impact on performance; when pulling an image consisting of multiple layers, layers can be pulled in parallel, and allows sharing layers between images (saving space).

For most use cases, multi-stage are a better alternative, as they give more fine-grained control over your build, and can take advantage of future optimizations in the builder. Refer to the use multi-stage builds (https://docs.docker.com/develop/develop-images/multistage-build/) section in the userguide for more information.

KNOWN LIMITATIONS

The `--squash` option has a number of known limitations:

- When squashing layers, the resulting image cannot take advantage of layer sharing with other images, and may use significantly more space. Sharing the base image is still supported.
- When using this option you may see significantly more space used due to storing two copies of the image, one for the build cache with all the cache layers in tact, and one for the squashed version.
- While squashing layers may produce smaller images, it may have a negative impact on performance, as a single layer takes longer to extract, and downloading a single layer cannot be parallelized.
- When attempting to squash an image that does not make changes to the filesystem (for example, the Dockerfile only contains `ENV` instructions), the squash step will fail (see issue #33823 (https://github.com/moby/moby/issues/33823)

PREREQUISITES

The example on this page is using experimental mode in Docker 1.13.

Experimental mode can be enabled by using the `--experimental` flag when starting the Docker daemon or setting `experimental: true` in the `daemon.json` configuration file.

By default, experimental mode is disabled. To see the current configuration, use the `docker version` command.

```
Server:
 Version:      1.13.1
 API version:  1.26 (minimum version 1.12)
 Go version:   go1.7.5
 Git commit:   092cba3
 Built:        Wed Feb  8 06:35:24 2017
 OS/Arch:      linux/amd64
 Experimental: false

 [...]
```

To enable experimental mode, users need to restart the docker daemon with the experimental flag enabled.

ENABLE DOCKER EXPERIMENTAL

Experimental features are now included in the standard Docker binaries as of version 1.13.0. For enabling experimental features, you need to start the Docker daemon with `--experimental` flag. You can also enable the daemon flag via /etc/docker/daemon.json. e.g.

```
{
    "experimental": true
}
```

Then make sure the experimental flag is enabled:

```
$ docker version -f '{{.Server.Experimental}}'
true
```

BUILD AN IMAGE WITH `--SQUASH` ARGUMENT

The following is an example of docker build with `--squash` argument

```
FROM busybox
RUN echo hello > /hello
RUN echo world >> /hello
RUN touch remove_me /remove_me
ENV HELLO world
RUN rm /remove_me
```

An image named `test` is built with `--squash` argument.

```
$ docker build --squash -t test .

[...]
```

If everything is right, the history will look like this:

```
$ docker history test

IMAGE               CREATED             CREATED BY                                      SIZE
            COMMENT
4e10cb5b4cac        3 seconds ago                                                       12 B
            merge sha256:88a7b0112a41826885df0e7072698006ee8f621c6ab99fca7fe9151d7b599702 to s
ha256:47bcc53f74dc94b1920f0b34f6036096526296767650f223433fe65c35f149eb
<missing>           5 minutes ago       /bin/sh -c rm /remove_me                        0 B
<missing>           5 minutes ago       /bin/sh -c #(nop) ENV HELLO=world               0 B
<missing>           5 minutes ago       /bin/sh -c touch remove_me /remove_me           0 B
<missing>           5 minutes ago       /bin/sh -c echo world >> /hello                 0 B
<missing>           6 minutes ago       /bin/sh -c echo hello > /hello                  0 B
<missing>           7 weeks ago         /bin/sh -c #(nop) CMD ["sh"]                    0 B
<missing>           7 weeks ago         /bin/sh -c #(nop) ADD file:47ca6e777c36a4cfff   1.113 M
B
```

We could find that all layer's name is `<missing>` , and there is a new layer with COMMENT `merge` .

Test the image, check for `/remove_me` being gone, make sure `hello\nworld` is in `/hello` , make sure the `HELLO` envvar's value is `world` .