Explaining Docker Image IDs

3 months ago

Docker Images · Docker · Docker Registries · Content Adressable · Image IDs · Image Layers

When Docker v1.10 came along, there was a fairly seismic change with the way the Docker Engine handles images. Whilst this was publicised well, and there was little impact on the general usage of Docker (image migration, aside), there were some UI changes which sparked some confusion. So, what was the change, and why does the `docker history` command show some IDs as `<missing>`?

```
$ docker history debian
IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT
1742affe03b5        10 days ago         /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
<missing>           10 days ago         /bin/sh -c #(nop) ADD file:5d8521419ad6cfb695   125.1 MB
```

Preamble

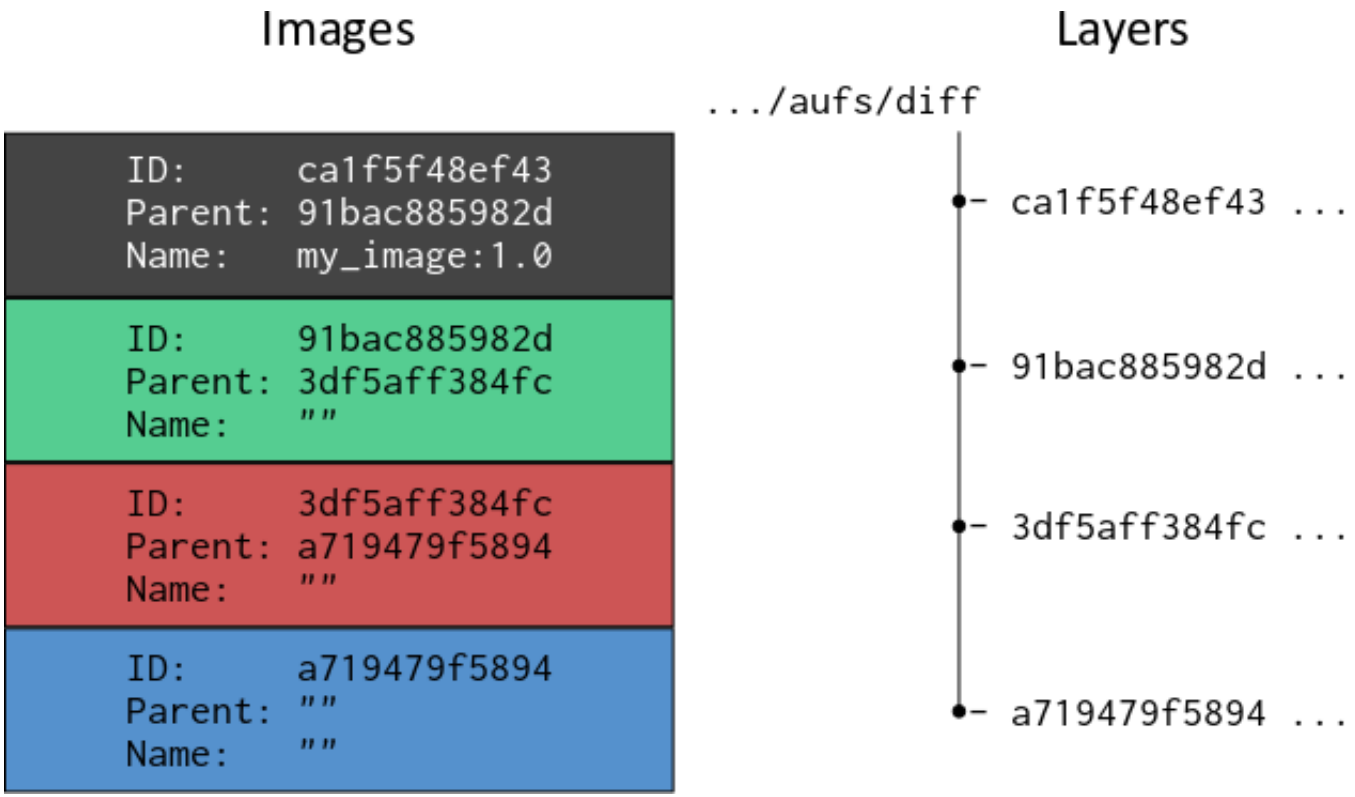
First, some background. A docker image is a read-only template for creating containers, and provides a filesystem based on an ordered union of multiple layers of files and directories, which can be shared with other images and containers. Sharing of image layers is a fundamental component of the Docker platform, and is possible through the implementation of a copy-on-write (COW) mechanism. During its lifetime, if a container needs to change a file from the read-only image that provides its filesystem, it copies the file up to its own private read-write layer before making the change.

A layer or 'diff' is created during the Docker image build process, and results when commands are run in a container,

which produce new or modified files and directories. These new or modified files and directories are 'committed' as a new layer. The output of the `docker history` command above shows that the `debian` image has two layers.

Historical Perspective

Historically (pre Docker v1.10), each time a new layer was created as a result of a commit action, Docker also created a corresponding image, which was identified by a randomly generated 256-bit UUID, usually referred to as an image ID (presented in the UI as either a short 12-digit hex string, or a long 64-digit hex string). Docker stored the layer contents in a directory with a name synonymous with the image ID. Internally, the image consisted of a configuration object, which held the characteristics of the image, including its ID, and the ID of the image's parent image. In this way, Docker was able to construct a filesystem for a container, with each image in turn referencing its parent and the corresponding layer content, until the base image was reached which had no parent. Optionally, each image could also be tagged with a meaningful name (e.g. `my_image:1.0`), but this was usually reserved for the leaf image. This is depicted in the diagram below:



Using the `docker inspect` command would yield:

```
$ docker inspect my_image:1.0

[

    {

        "Id": "ca1f5f48ef431c0818d5e8797dfe707557bdc728fe7c3027c75de18f934a3b76",

        "Parent": "91bac885982d2d564c0e1869e8b8827c435eead714c06d4c670aaae616c1542c"

        ...

        ...
```

This method served Docker well for a sustained period, but over time was perceived to be sub-optimal for a variety of reasons. One of the big drivers for change, came from the lack of a means of detecting whether an image's contents had been tampered with during a push to or pull from a registry, such as the Docker Hub. This led to robust criticism from the community at large, and led to a series of changes, culminating in content addressable IDs.

Content Addressable IDs

Since Docker v1.10, generally, images and layers are no longer synonymous. Instead, an image directly references one or more layers that eventually contribute to a derived container's filesystem.

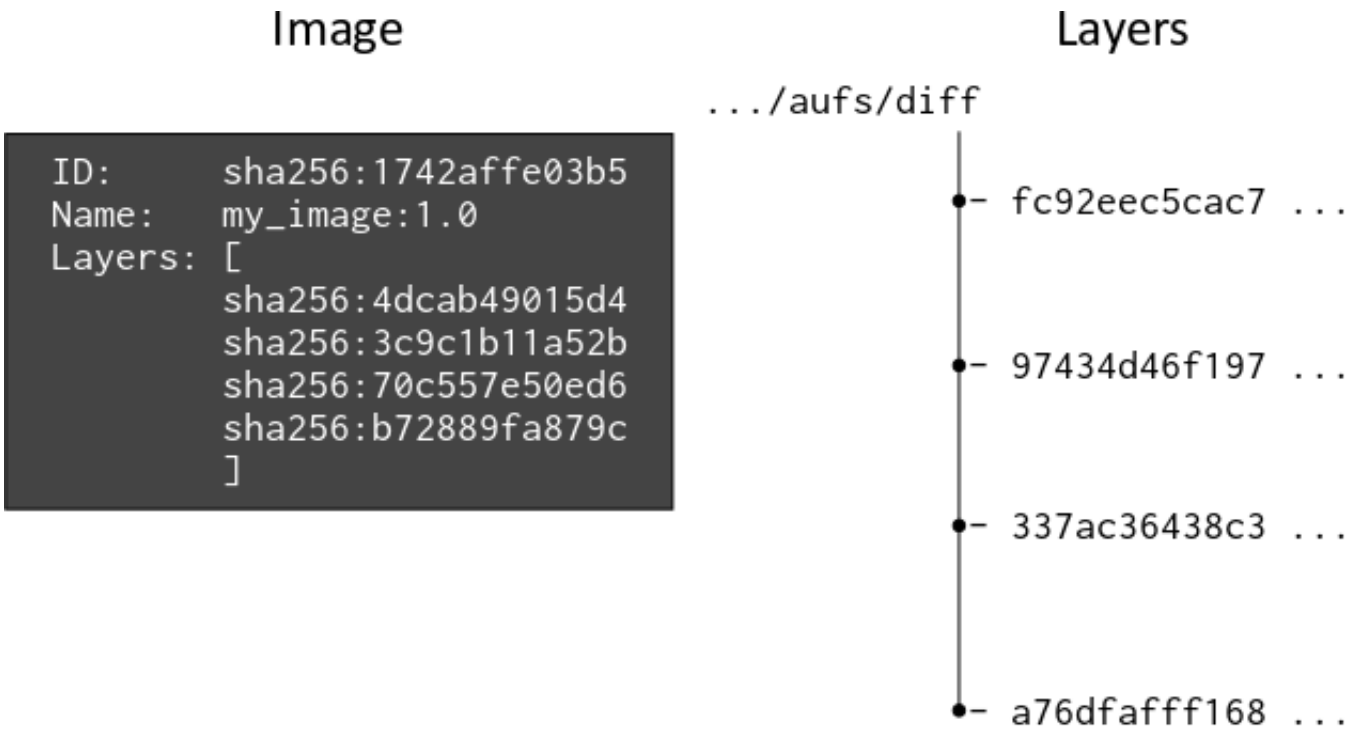
Layers are now identified by a digest, which takes the form `algorithm:hex`; for example:

```
sha256:fc92eec5cac70b0c324cec2933cd7db1c0eae7c9e2649e42d02e77eb6da0d15f
```

The hex element is calculated by applying the algorithm (SHA256) to a layer's content. If the content changes, then the computed digest will also change, meaning that Docker can check the retrieved contents of a layer with its published digest in order to verify its content. Layers have no notion of an image or of belonging to an image, they are merely collections of files and directories.

A Docker image now consists of a configuration object, which (amongst other things) contains an ordered list of layer digests, which enables the Docker Engine to assemble a container's filesystem with reference to layer digests rather than parent images. The image ID is also a digest, and is a computed SHA256 hash of the image configuration object, which contains the digests of the layers that contribute to the image's filesystem definition. The following diagram depicts the

relationship between image and layers post Docker v1.10:



The digests for the image and layers have been shortened for readability.

The `diff` directory for storing the layer content, is now named after a randomly generated 'cache ID', and the Docker Engine maintains the link between the layer and its cache ID, so that it knows where to locate the layer's content on disk.

So, when a Docker image is pulled from a registry, and the `docker history` command is used to reveal its contents, the output provides something similar to:

```
$ docker history swarm
```

```
IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT
c54bba046158        9 days ago          /bin/sh -c #(nop) CMD ["--help"]                0 B
<missing>           9 days ago          /bin/sh -c #(nop) ENTRYPOINT &{["/swarm"]}      0 B
<missing>           9 days ago          /bin/sh -c #(nop) VOLUME [/.swarm]              0 B
<missing>           9 days ago          /bin/sh -c #(nop) EXPOSE 2375/tcp               0 B
<missing>           9 days ago          /bin/sh -c #(nop) ENV SWARM_HOST=:2375          0 B
<missing>           9 days ago          /bin/sh -c #(nop) COPY dir:b76b2255a3b423981a   0 B
<missing>           9 days ago          /bin/sh -c #(nop) COPY file:5acf949e76228329d   277.2 kB
<missing>           9 days ago          /bin/sh -c #(nop) COPY file:a2157cec2320f541a   19.06 MB
```

The command provides detail about the image and the layers it is composed of. The `<missing>` value in the `IMAGE` field for all but one of the layers of the image, is misleading and a little unfortunate. It conveys the suggestion of an error, but there is no error as layers are no longer synonymous with a corresponding image and ID. I think it would have been more appropriate to have left the field blank. Also, the image ID appears to be associated with the uppermost layer, but in fact, the image ID doesn't 'belong' to any of the layers. Rather, the layers collectively belong to the image, and provide its filesystem definition.

Locally Built Images

Whilst this narrative for content addressable images holds true for all Docker images post Docker v1.10, locally built images on a Docker host are treated slightly differently. The generic content of an image built locally remains the same - it is a configuration object containing configuration items, including an ordered list of layer digests.

However, when a layer is committed during an image build on a local Docker host, an 'intermediate' image is created at

the same time. Just like all other images, it has a configuration item which is a list of the layer digests that are to be incorporated as part of the image, and its ID or digest contains a hash of the configuration object. Intermediate images aren't tagged with a name, but, they do have a 'Parent' key, which contains the ID of the parent image.

The purpose of the intermediate images and the reference to parent images, is to facilitate the use of Docker's build cache. The build cache is another important feature of the Docker platform, and is used to help the Docker Engine make use of pre-existing layer content, rather than regenerating the content needlessly for an identical build command. It makes the build process more efficient. When an image is built locally, the `docker history` command might provide output similar to the following:

```
$ docker history jbloggs/my_image:latest
IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT
26cca5b0c787        52 seconds ago      /bin/sh -c #(nop) CMD ["/bin/sh" "-c" "/bin/b   0 B
97e47fb9e0a6        52 seconds ago      /bin/sh -c apt-get update &&     apt-get inst   16.98 MB
1742affe03b5        13 days ago         /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
<missing>           13 days ago         /bin/sh -c #(nop) ADD file:5d8521419ad6cfb695   125.1 MB
```

In this example, the top two layers are created during the local image build, whilst the bottom layers came from the base image for the build (e.g. Dockerfile instruction `FROM debian`). We can use the `docker inspect` command to review the layer digests associated with the image:

```
$ docker inspect jboggs/my_image:latest
```

```
[
    {
        ...
        ...
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:4dcab49015d47e8f300ec33400a02cebc7b54cadd09c37e49eccbc655279da90",
                "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef",
                "sha256:f22bfbc1df820c465d94215e45bf9b1f0ae0fe3435a90dc5296a4b55712f46e7"
            ]
        }
    }
]
```


The `docker history` command shows the image as having four layers, but `docker inspect` suggests just three layers. This is because the two `CMD` instructions produce metadata for the image, don't add any content, and therefore the 'diff' is empty. The digest `5f70bf18a08a` is the SHA256 hash of an empty layer, and is shared by both of the layers in question.

When a locally built image is pushed to a registry, it is only the leaf image that is uploaded along with its constituent layers, and a subsequent pull by another Docker host will not yield any intermediate parent images. This is because once the image is made available to other potential users on different Docker hosts via a registry, it effectively becomes read-only, and the components that support the build cache are no longer required. Instead of the image ID, `<missing>` is inserted into the history output in its place.

Pushing the image to a registry might yield:

```
$ docker push jbloggs/my_image:latest
The push refers to a repository [docker.io/jbloggs/my_image]
f22bfbc1df82: Pushed
5f70bf18a086: Layer already exists
4dcab49015d4: Layer already exists
latest: digest: sha256:7f63e3661b1377e2658e458ac1ff6d5e0079f0cfd9ff2830786d1b45ae1bb820 size: 3147
```

In this example, only one layer has been pushed, as two of the layers already exist in the registry, referenced by one or more other images which use the same content.

A Final Twist

The digests that Docker uses for layer 'diffs' on a Docker host, contain the sha256 hash of the tar archived content of the diff. Before the layer is uploaded to a registry as part of a push, it is compressed for bandwidth efficiency. A manifest is also created to describe the contents of the image, and it contains the digests of the **compressed** layer content. Consequently, the digests for the layers in the manifest are different to those generated in their uncompressed state. The manifest is also pushed to the registry.

The digest of a compressed layer diff can be referred to as a 'distribution digest', whilst the digest for the uncompressed layer diff can be referred to as a 'content digest'. Hence, when we pull our example image on a different Docker host, the `docker pull` command gives the following output:

```
$ docker pull jbloggs/my_image
Using default tag: latest
latest: Pulling from jbloggs/my_image

51f5c6a04d83: Pull complete
a3ed95caeb02: Pull complete
9a246d793396: Pull complete
Digest: sha256:7f63e3661b1377e2658e458ac1ff6d5e0079f0cfd9ff2830786d1b45ae1bb820
Status: Downloaded newer image for jbloggs/my_image:latest
```

The distribution digests in the output of the `docker pull` command, are very different to the digests reported by the `docker push` command. But, the pull will decompress the layers, and the output of a `docker inspect` command will provide the familiar content digests that we saw after the image build.

Summary

Following the changes to image and layer handling in Docker v1.10:

- A Docker image provides a filesystem for a derived container based on the references it stores to layer diffs
- Layer diffs are referenced using a digest, which contains an SHA256 hash of an archive of the diff's contents
- A Docker image's ID is a digest, which contains an SHA256 hash of the image's JSON configuration object
- Docker creates intermediate images during a local image build, for the purposes of maintaining a build cache
- An image manifest is created and pushed to a Docker registry when an image is pushed

- An image manifest contains digests of the image's layers, which contain the SHA256 hashes of the compressed, archived diff contents

Nigel Brown

IT Professional

Woodbridge, United Kingdom

Share this post