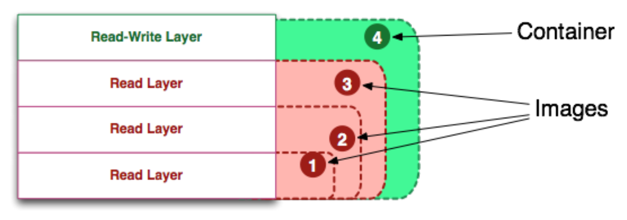
Probably Done Before

Visualizing Docker Containers and Images

This post is meant as a Docker **102**-level post. If you are unaware of what Docker is, or don't know how it compares to virtual machines or to configuration management tools, then this post might be a bit too advanced at this time.

This post hopes to aid those struggling to internalize the docker command-line, specifically with knowing the exact difference between a **container** and an **image.** More specifically, this post shall differentiate a simple **container** from a **running container**.

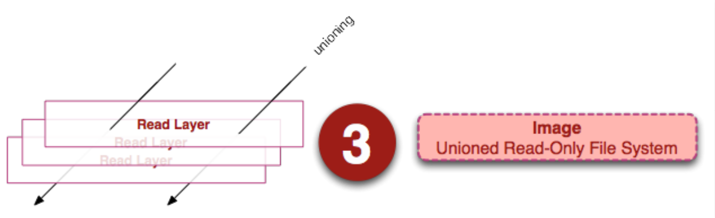


I do this by taking a look at some of the underlying details, namely the layers of the union file system. This was a process I undertook for myself in the past few weeks, as I am relatively new to the docker technology and have found the docker command-lines difficult to internalize.

Tangent: In my opinion, understanding how a technology works under the hood is the best way to achieve learning speed and to build confidence that you are using the tool in the correct way. Often a technology is released with a certain breathless and hype that make it difficult to really understand appropriate usage patterns. More specifically, technology releases often develop an abstraction model that can invent new terminologies and metaphors that might be useful at first, but make it harder to develop mastery in latter stages.

A good example of this is Git. I could not gain traction with Git until I understood its underlying model, including trees, blobs, commits,tags, tree-ish, etc. I had written about this before in a previous post, and still remain convinced that people who don't understand the internals of Git cannot have true mastery of the tool.

Image Definition
The first visual I present is that of an **image**, shown below with two different visuals. It is defined as the "union view" of a stack of **read-only layers**.



On the left we see a stack of read-layers. These layers are internal implementation details only, and are accessible outside of running containers in the host's file system. Importantly, they are read-only (or immutable) but capture the changes (deltas) made to the layers below. Each layer may have one parent, which itself may have a parent, etc. The top-level layer may be read by a union-ing file system (AUFS on my docker implementation) to present a single cohesive view of all the changes as *one* read-only file system. We see this "union view" on the right.

If you want to see these layers in their glory, you might find them in different locations on your host's files system. These layers will not be viewable from within a running container directly. In my docker's host system I can see them at `/var/lib/docker` in a subdirectory called `aufs`.

```
# sudo tree -L 1 /var/lib/docker/

/var/lib/docker/

├── aufs

├── containers

├── graph

├── init

├── linkgraph.db

├── repositories-aufs

├── tmp

├── trust

└── volumes


7 directories, 2 files
```

Container Definition
A container is defined as a "union view" of a stack of layers the top of which is a read-write layer.

Previous Posts

- ► 2010 (9)
- ► 2011 (3)
- ► 2013 (2)
- ► 2014 (5)
- ▼ 2015 (1)
  - ▼ October (1)
    - Visualizing Docker Containers and Images

About Me

Daniel Eklund
Boston, MA

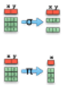
I am 20 year programmer/consultant with a background in enterprise software engineering and big data. This blog is my place to keep up with my love of functional programming languages, and foundational mathematics. Recently, I've been creating a site from Erlang, Elixir and Haskell, called Wordadoplicus -- a site for old-school boggle-type real-time word games.

View my complete profile

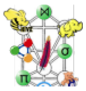
Popular Posts

Why the Heck is Git so Hard?
Why the Heck is Git so Hard? The Places Model™ ... ok maybe not hard , but complicated (which is not a bad thing) This post is ai...

Visualizing Docker Containers and Images
This post is meant as a Docker 102 -level post. If you are unaware of what Docker is, or don't know how it compares to virtual machi...

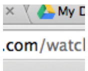


What the Heck are Algebraic Data Types? (for Programmers)
This post is meant to be a gentle introduction to Algebraic Data Types. Some of you may be asking why you should learn Algebraic Data Typ...



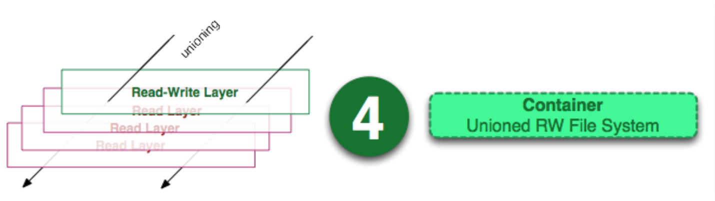
The Occultation of Relations and Logic: Exposing the Hidden Meaning from within Shadows and Unix Command Lines
Here is a question and answer stolen from another website : " I have [an] nginx log file, and I want to find out market sha...

What the Heck is a Relation? From Tables to Cartesian Products to Logic
Awww... I have given several talks and courses on Apache Hive and Pig and other new-ish 'Big Data' languages built upon Hadoop an...



What the Heck is Base64 Encoding really?
Also, Why most implementations of Crockford-base32 encoding are probably incorrect TLDR : The term base64 encoding is an overload...

Animating the Git Branching Model: Part 1
I've been trying to learn Apple Motion 5. In my day job, I draw



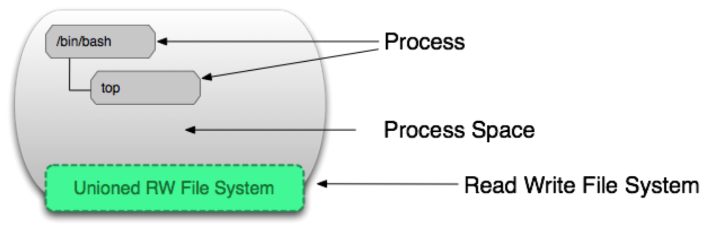
I show this visual above, and you will note it is nearly the same thing as an image, except that the top layer is read-write. At this point, some of you might notice that this definition says nothing about whether this container is running, and this is on purpose. It was this discovery in particular that cleared up a lot of confusion I had up to this point.

Takeaway: A container is defined *only* as a read-write layer atop an image (of read-only layers itself). **It does not have to be running**.

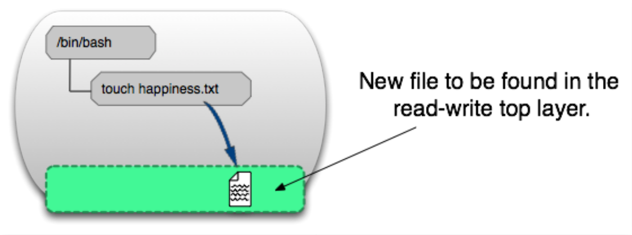
So if we want to discuss containers running, we need to define a **running container**.

Running Container Definition

A ***running* container** is defined as a read-write "union view" and the the isolated process-space and processes within. The below visual shows the read-write container surrounded by this process-space.



It is this act of isolation atop the file system effected by kernel-level technologies like cgroups, namespaces, etc that have made docker such a promising technology. The processes within this process-space may change, delete or create files within the "union view" file that will be captured in the read-write layer. I show this in the visual below:

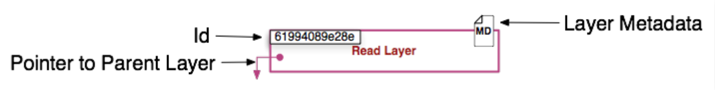


To see this at work run the following command: **`docker run ubuntu touch happiness.txt`**. You will then be able to see the new file in the read-write layer of the host system, even though there is no longer a running container (note, run this in your host system, not a container):

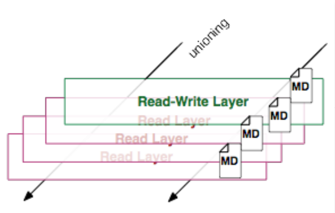
```
# find / -name happiness.txt
/var/lib/docker/aufs/diff/860a7b...889/happiness.txt
```

Image Layer Definition

Finally, to tie up some loose ends, we should define an image layer. The below image shows an image layer and makes us realize that *a layer is not just the changes to the file system*.



The metadata is additional information about the layer that allows docker to capture runtime and build-time information, but also hierarchical information on a layer's parent. *Both* read and read-write layers contain this metadata.



Additionally, as we have mentioned before, each layer contains a pointer to a parent layer using the Id (here, the parent layers are below). If a layer does not point to a parent layer, then it is at the bottom of the stack.

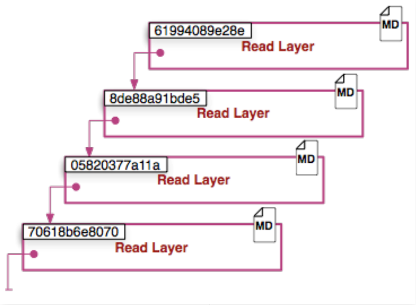
a lot of architecture diagrams to communicate with clients and with my ...

Gotcha with Bitstrings in Erlang: Naked Chevron Numerals are Decimal

Some Confusing Erlang Bitstring Syntax   or how "A" is 01000001 in decimal and binary TLDR Summary <<0100000...

Erlang Radix Syntax, and Lambda as the Ultimate Meaning

Follow Up to "Gotcha with Bitstrings"   How to ACTUALLY denote binary numbers in Erlang After I put the previous post together, I...



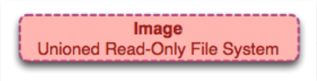
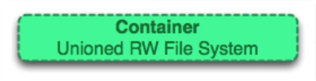
Metadata Location:
At this time (and I'm fully aware that the docker developers could change the implementation), the metadata for an **image** (read-only) layer can be found in a file called "`json`" within `/var/lib/docker/graph` at the id of the particular layer:
`/var/lib/docker/graph/e809f156dc985.../json`
where "`e809f156dc985...`" is the elided id of the layer.

The metadata for a **container** seems to be broken into many files, but more or less is found in `/var/lib/docker/containers/<id>` where <id>is the id of the read-write layer. The files in this directory contain more of the run-time metadata needed to expose a container to the outside world: networking, naming, logs, etc.

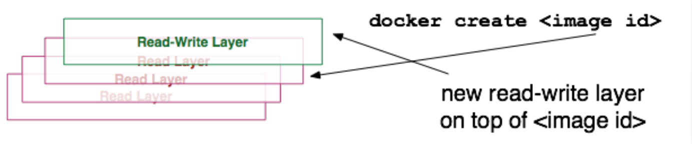
Tying It All Together

Now, let's look at the commands in the light of these visual metaphors and implementation details.

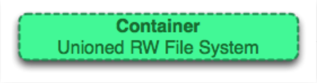
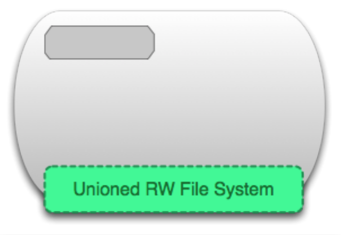
`docker create <image-id>`

Input (if applicable)	**Output** (if applicable)
	

The 'docker create' command adds a read-write layer to the top stack based on the image id. It *does not* run this container.

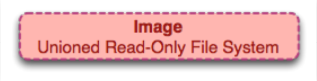
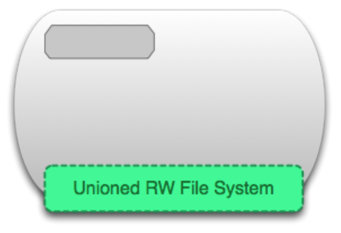


`docker start <container-id>`

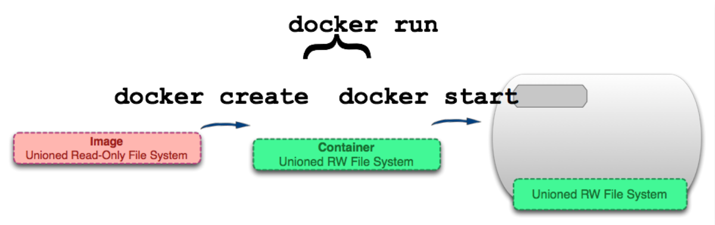
Input (if applicable)	**Output** (if applicable)
	

The command 'docker start' creates a process space around the union view of the container's layers. There can only be one process space per container.

`docker run <image-id>`

Input (if applicable)	**Output** (if applicable)
	

One of the first questions people ask (myself included) is "What is the difference between 'docker start' and 'docker run'. You might argue that the entire point of this post is to explain the subtleties in this distinction.

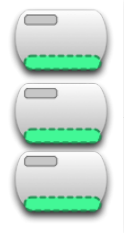


As we can see, the docker run command starts with an **image**, creates a **container**, and starts the container (turning it into a **running container**). It is very much a convenience, and hides the details of two commands.

Tangent: Continuing with the aforementioned similarity to understanding the Git system, I consider the 'docker run' command to be similar to the 'git pull'. Like 'git pull' (which is a combination of 'git fetch' and 'git merge') the 'docker run' is a combination of two underlying commands that have meaning and power on their own.

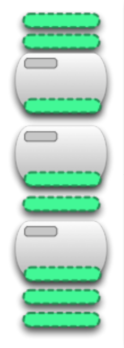
In this sense it is certainly convenient, but potentially apt to create misunderstandings.

`docker ps`

Input (if applicable)	**Output** (if applicable)
your host system	

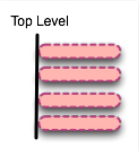
The command 'docker ps' lists out the inventory of **running containers** on your system. This is a very important filter that hides the fact that containers exist in a non-running state. To see non-running containers too, we need to use the next command.

`docker ps -a`

Input (if applicable)	**Output** (if applicable)
your host system	

The command 'docker ps -a' where the 'a' is short for 'all' lists out all the containers on your system, whether stopped or running.

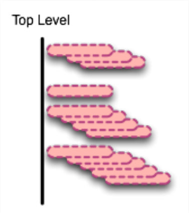
`docker images`

Input (if applicable)	**Output** (if applicable)
you host system	

The 'docker images' command lists out the inventor of top-level images on your system. **Effectively there is nothing to distinguish an image from a read-only layer**. Only those images that have containers attached to them or that have been pulled are considered top-level. This distinction is for convenience as there are may be many hidden layers beneath each top-level read-only layer.

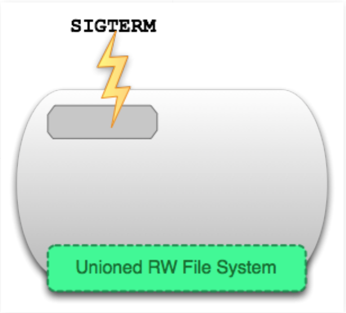
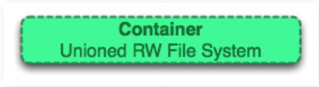
`docker images -a`

Input (if applicable)	**Output** (if applicable)

you host system	

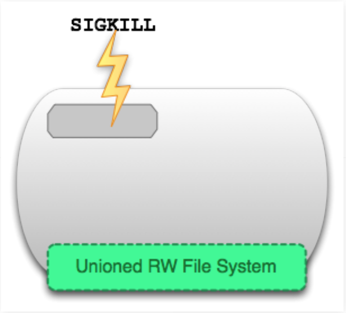
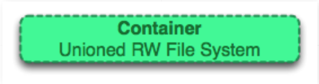
This command 'docker images -a' shows all the images on your system. This is exactly the same as showing all the read-only layers on the system. If you want to see the layers below one image-id, you should use the 'docker history' command discussed below.

docker stop <container-id>

Input (if applicable)	**Output** (if applicable)
	

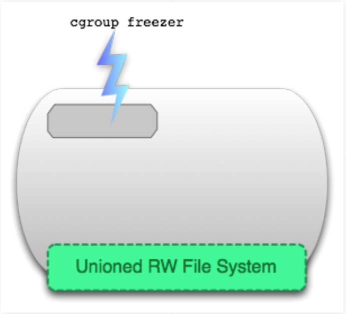
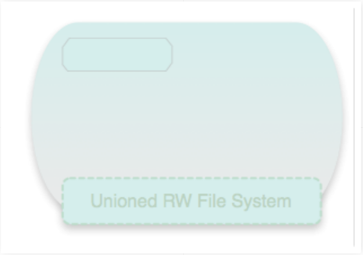
The command 'docker stop' issues a SIGTERM to a running container which politely stops all the processes in that process-space. What results is a normal, but non-running, container.

docker kill <container-id>

Input (if applicable)	**Output** (if applicable)
	

The command 'docker kill' issues a non-polite SIGKILL command to all the processes in a running container. ~~This is the same thing as hitting Control-C in your shell~~. (EDIT: Control-C sends a SIGINT)

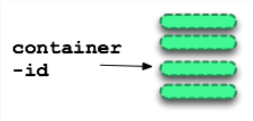
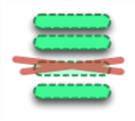
docker pause <container-id>

Input (if applicable)	**Output** (if applicable)
	

Unlike 'docker stop' and 'docker kill' which send actual UNIX signals to a running process, the command 'docker pause' uses a special cgroups feature to freeze/pause a running process-space. The rationale can be found here: https://www.kernel.org/doc/Documentation/cgroups/freezer-subsystem.txt, but the short of it is that sending a Control-Z (SIGTSTP) is not transparent enough to the processes within the process-space to truly allow all of them to be frozen.

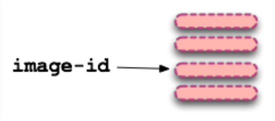
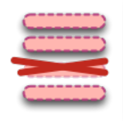
docker rm <container-id>

Input (if applicable)	**Output** (if applicable)

The command 'docker rm' removes the read-write layer that defines a container from your host system. It must be run on stopped containers. It effectively deletes files.

docker rmi <image-id>

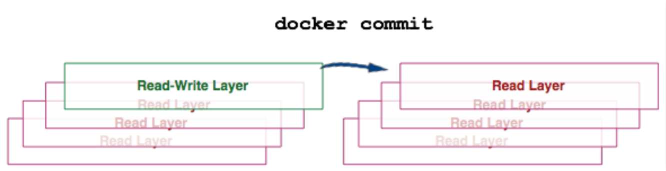
Input (if applicable)	**Output** (if applicable)
	

The command 'docker rmi' removes the read-layer that defines a "union view" of an image. It removes this image from your host, though the image may still be found from the repository from which you issued a 'docker pull'. You can only use 'docker rmi' on top-level layers (or images), and not on intermediate read-only layers (unless you use -f for 'force').

docker commit <container-id>

Input (if applicable)	**Output** (if applicable)
or	

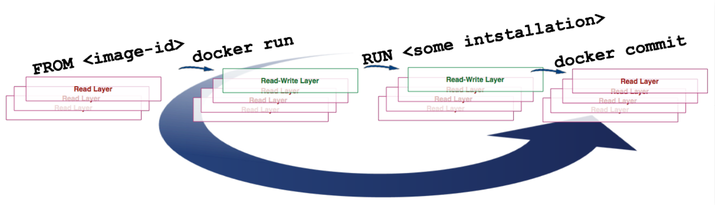
The command 'docker commit' takes a container's top-level read-write layer and burns it into a read-only layer. This effectively turns a container (whether running or stopped) into an immutable image.



docker build

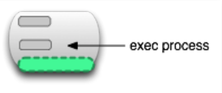
Input (if applicable)	**Output** (if applicable)
Dockerfile: and a 	 with many more layers added atop.

The 'docker build' command is an interesting one as it iteratively runs multiple commands at once.



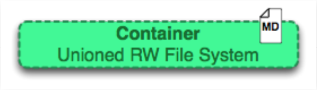
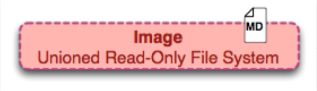
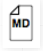
We see this in the above visual which shows how the build command uses the FROM directive in the Dockerfile file as the starting image and iteratively 1) runs (create and start) 2) modifies and 3) commits. At each step in the iteration a new layer is created. Many new layers may be created from running a 'docker build'.

docker exec <running-container-id>

Input (if applicable)	**Output** (if applicable)
	

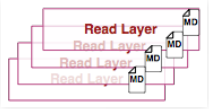
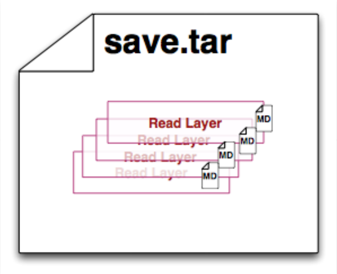
The 'docker exec' command runs on a running container and executes a process in that running container's process space.

docker inspect <container-id> or <image-id>

Input (if applicable)	**Output** (if applicable)
 or 	

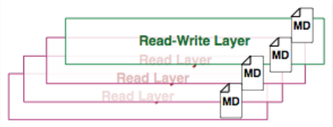
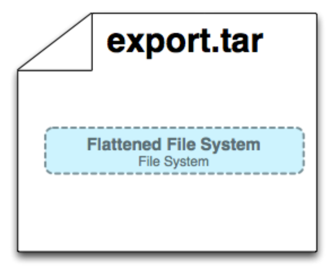
The command 'docker inspect' fetches the metadata that has been associated with the top-layer of the container or image.

`docker save <image-id>`

Input (if applicable)	**Output** (if applicable)
	 save.tar Read Layer Read Layer Read Layer Read Layer MD

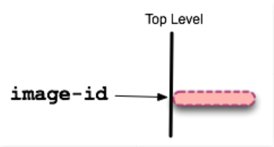
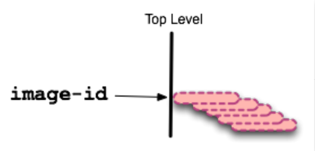
The command 'docker save' creates a single tar file that can be used to import on a different host system. Unlike the 'export' command, it saves the individual layers with all their metadata. This command can only be run on an image.

`docker export <container-id>`

Input (if applicable)	**Output** (if applicable)
 Read-Write Layer Read Layer Read Layer Read Layer MD	

The 'docker export' command creates a tar file of the contents of the "union view" and flattens it for consumption for non-Docker usages. This command removes the metadata and the layers. This command can only be run on containers.

`docker history <image-id>`

Input (if applicable)	**Output** (if applicable)
	

The 'docker history' command takes an image-id and recursively prints out the read-only layers (which are themselves images) that are ancestors of the input image-id.

Conclusion

I hope you enjoyed this visualization of containers and images. There are many other commands (pull, search, restart, attach, etc) which may or may not relate to these metaphors. I believe though that the great majority of docker's primary commands can be easier understood with this effort. I am only two weeks into learning docker, so if I missed a point or something can be better explained, please drop a comment.

Posted by Daniel Eklund on Monday, October 26, 2015

39 comments:

anant October 26, 2015 at 9:26 PM

Great Post!

Reply

Greg Bray October 26, 2015 at 11:53 PM

Quick question... Can I edit files in the unioned RW filesystem directly on the host filesystem? Like your happiness.txt example above, if I change the contents on the host while the container is running (or stopped if the container locks the file) will the container see the new contents?

Reply

Replies

Dhawal Yogesh Bhanushali October 27, 2015 at 3:47 AM

yes. If you can ls them, then they are at your mercy to not edit them :-) You edit them with run.

Reply

Greg Bray October 27, 2015 at 12:11 AM

Also I noticed that you don't have a license listed on your blog content. By default that makes your posts and images fully copyrighted, which is fine if that is your intent, but a more permissive license would let others share or improve your work while still following your wishes (non commercial, include attribution/links, etc)

You might want to look at http://creativecommons.org which can help you choose a license that would allow your images or words to be re-used or modified while still respecting whatever limits you want to impose.

Keep up the great work!

Reply

Anonymous October 27, 2015 at 2:58 AM

As a newcomer to docker, I had some trouble with the distinction between create, run and start. Thanks to your excellent images it all became clear to me. Thank you!

Reply

Sharas October 27, 2015 at 3:19 AM

good stuff. sometimes software tools are too pretentious. makes it hard to learn. Learning by understanding is much more powerful than learning by rot.

Reply

Unknown October 27, 2015 at 6:49 AM

I second this. I'd like to link to your article but also use some of the images with your permission.

This material is very helpful. Thanks for putting it together.

Reply

Ahmet Alp Balkan October 27, 2015 at 11:44 AM

Great post, thanks for all the figures! I may have one point to correct: You said “aufs on my docker implementation”, I think you might want to say “... my docker setup” there.

Another correction. In `rmi` you said -f will remove intermediate layers. I don't think it would remove an intermediate layer like that (on my machine it says “conflict, $LAYER wasn't deleted”). On the other hand if an image's topmost layer is an intermediate layer of some other image, you can remove the earlier image (such as:d docker rmi -f ubuntu) and it still won't delete the layer, it will just say:

```
$ docker rmi -f busybox
Untagged: busybox:latest
```

but if you have no other images deriving from topmost layer of busybox you will get:

```
docker rmi -f busybox
Untagged: busybox:latest
Deleted: 3d5bcd78e074f6f77b820bf4c6db0e05d522e24c855f3c2a3bbf3b1c8f967ba8
Deleted: bf0f46991aed1fa45508683e740601df23a2395c47f793883e4b0160ab906c1e
```

and that's when layers will get deleted.

Reply

Anonymous October 28, 2015 at 1:30 PM

Nice post and probably docker should consider adding it to their document! I struggled with the understanding of images/containers when I first started with docker.

Reply

Bala Paranj October 28, 2015 at 2:31 PM

Very useful diagrams. Thanks.

Reply

Unknown October 28, 2015 at 9:07 PM

Great! I've cost so much time to study details between image and container in docker. If I found this post firstly, I can save such time.

Reply

matt October 28, 2015 at 10:05 PM

Wow what a great post. I finally get it! Off to read your post on git now :)

Reply

raliski October 29, 2015 at 3:34 AM

Wow! This is what I was looking for since about a year or so..Thanks!

Reply

myopia October 29, 2015 at 1:23 PM

A very useful contribution.

Would you please tell us what tools you used to draw these things. Thanks!

Reply

afolarin October 31, 2015 at 1:14 AM

Excellent post

Reply

Davin Tryon November 30, 2015 at 9:27 AM

Very useful! Thank you.

Reply

Rahul Bansal January 24, 2016 at 10:58 PM

Thanks a ton for amazing post. :-)

I now understand dockers much better than before.

Reply

SANTHOSH SAM January 26, 2016 at 2:49 AM

Hi Some images are not loading i think some content Delivery Issue, Kindly check.

Thanks

Reply

Oriol Rius January 30, 2016 at 1:38 AM

This is the best description that I read about how docker works, simple and clear. Straight to the point.

Thanks!

Reply

volkan Tufekci February 4, 2016 at 11:01 PM

Well done! Thank you

Reply

Anonymous February 27, 2016 at 8:16 PM

Great Post ! One question I had while reading was what is the Ctrl + <> for SIGKILL.

Reply

john harris March 1, 2016 at 11:31 PM

This comment has been removed by a blog administrator.

Reply

Chris Wolfgang March 21, 2016 at 8:53 AM

Hi, Daniel! I'm the editor of Codeship's blog, and I recently ran across this post. Would you be interested in letting us republish it on the Codeship blog? Very occasionally, we seek permission to republish an author's work that would be of particular interest to our audience. Of course, we maintain your original post as the canonical URL on our blog. I'd be honored if you'd consider it.

Feel free to email me if you've got any questions at all! chris dot wolfgang at codeship dot com.
Thanks!

Reply

John April 25, 2016 at 7:59 PM

Thanks for the sharing!

Reply

cecchisandrone May 13, 2016 at 6:15 AM

Thanks...great post!

Reply

任喜军 July 4, 2016 at 7:29 PM

Very nice post and helpful to me, thanks

Reply

Matt Sun August 21, 2016 at 12:15 AM

This article is fxxking awesome! Better than all the articles I've ever seen before! Many thanks!

Reply

Anonymous September 6, 2016 at 8:27 AM

tremendous, thank you

Reply

Anonymous November 11, 2016 at 5:26 PM

Great post!!

Reply

Unknown February 28, 2017 at 12:23 PM

Thank you! This is what I was looking for when I did my google search. Really Good!

Reply

Anonymous March 8, 2018 at 7:00 AM

Introduce basic but essential concepts in a easy-to-understand way. If you are new to the docker world, this article is must reading.

Reply

littlekey July 9, 2018 at 6:53 AM

excellent!

Reply

mahesh chemmala October 1, 2018 at 12:49 AM

Thank you for posting the valuable information about the Docker Blogs .And every people easily understand about your posting, and I am learning a lot of things from your posts,Keep it up.
DevOps Online Training

Reply

Unknown November 26, 2018 at 10:06 PM

That is the best docker-related post I've ever read. Good job!

Reply

dockerdevops January 24, 2019 at 10:42 PM

Thanks for sharing this information
<a href="https://www.visualpath.in/DevOps-docker-kubernetes-training.html"> Docker and Kubernetes Training in Hyderabad </a>

Reply

Imran Sayed January 28, 2019 at 1:27 AM

Good explanation and visual representation of images, containers and commands

Reply

dockerdevops February 12, 2019 at 10:47 PM

Very useful Blog.. Thanks for sharing....!!

Docker and Kubernetes Training

Reply

Unknown March 1, 2019 at 1:23 AM

thanks

Reply

Naveen March 15, 2019 at 11:58 PM

Thank you for excellent article.

Please refer below if you are looking for best project center in coimbatore

Java training in coimbatore
soft skill training in coimbatore
final year projects in coimbatore
Spoken English Training in coimbatore
final year projects for CSE in coimbatore
final year projects for IT in coimbatore
final year projects for ECE in coimbatore
final year projects for EEE in coimbatore
final year projects for Mechanical in coimbatore
final year projects for Instrumentation in coimbatore

Reply

Enter your comment...

Comment as: Google Account

Publish Preview

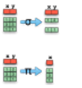
Home Older Post

Subscribe to: Post Comments (Atom)

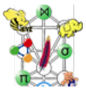
Popular Posts

Why the Heck is Git so Hard?
Why the Heck is Git so Hard? The Places Model™   ... ok maybe not hard , but complicated (which is not a bad thing) This post is ai...

Visualizing Docker Containers and Images
This post is meant as a Docker 102 -level post. If you are unaware of what Docker is, or don't know how it compares to virtual machi...

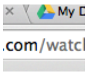


What the Heck are Algebraic Data Types? (for Programmers)
This post is meant to be a gentle introduction to Algebraic Data Types. Some of you may be asking why you should learn Algebraic Data Typ...



The Occultation of Relations and Logic: Exposing the Hidden Meaning from within Shadows and Unix Command Lines
Here is a question and answer stolen from another website :   " I have [an] nginx log file, and I want to find out market sha...

What the Heck is a Relation? From Tables to Cartesian Products to Logic
Awww... I have given several talks and courses on Apache Hive and Pig and other new-ish 'Big Data' languages built upon Hadoop an...



What the Heck is Base64 Encoding really?
Also, Why most implementations of Crockford-base32 encoding are probably incorrect TLDR : The term base64 encoding is an overload...

Animating the Git Branching Model: Part 1
I've been trying to learn Apple Motion 5. In my day job, I draw a lot of architecture diagrams to communicate with clients and with my ...

Gotcha with Bitstrings in Erlang: Naked Chevron Numerals are Decimal
Some Confusing Erlang Bitstring Syntax   or how "A" is 01000001 in decimal and binary TLDR Summary <<0100000...

Erlang Radix Syntax, and Lambda as the Ultimate Meaning
Follow Up to "Gotcha with Bitstrings"   How to ACTUALLY denote binary numbers in Erlang After I put the previous post together, I...

Simple theme. Powered by Blogger.