All my Docker collections

- Docker Intro & Installation
- Docker Image Building
- Docker Advanced Volumes
- Docker for Hadoop

WL

Others, node LTS

It's based on devenv-sid.

prep node 12

```
ccd ~/l/bld/docker/wrkenv-std
cp wrkenv-work.json wrkenv-node.json
```

```
ccd ~/l/bld/docker
cp -a devenv-sid/ devenv-node
```

Build

```
ccd ~/l/bld/docker
```

Do “sfxpt/debian:wrk, work env base > Build” first, with

```
docker build --rm -t sfxpt/debian:wrk --build-arg PACKAGES="" build
```

```
# change xterm to at least twice wide -- 160 columns
( cd wrkenv-std; time packer build -only=docker -var senv_user_uid=`id -u` -var senv_user_gid=1001 wrkenv-cust.json )
```

```
# still under ~/l/bld/docker, build sfxpt/wrkenv-std:nodebase
( cd wrkenv-std; time packer build -only=docker wrkenv-node.json )
```

# still under ~/l/bld/docker, as per sfxpt/debian:sid

```
docker build --rm -t sfxpt/wrkenv-std:nodets -f devenv-node/Dockerfile .
```

Run

```
        docker rm -f wrkenv-nodeLTS
docker run -ti --name wrkenv-nodeLTS --hostname wrkenv-std -v /tmp:/tmp -v /lfs:/lfs -v /export/build:/export/build -v ~/l:/home/me/l --user me sfxpt/wrkenv-std:nodets /bin/bash -i
docker ps -a
# continue, in git-bash!
docker start -ai wrkenv-nodeLTS
```

Docker build

http://www.docker.io/learn/dockerfile/level1/

Once you have your Dockerfile ready, you can use the `docker build` command to create your image from it. There are different ways to use this command:

1. If your Dockerfile is in your current directory you can use `docker build .`
2. From stdin. For instance: `docker build - < Dockerfile`
3. From GitHub. For instance: `docker build github.com/creack/docker-firefox`
4. In this last example, docker will clone the GitHub repository and use it as context. The Dockerfile at the root of the repository will be used to create the image.

Delete all stopped docker containers

1. **Take out the garbage**
    - Here is how to delete all stopped containers (useful if you need to re-run named containers):
    - `sudo ls /var/lib/docker/containers/`
    - `docker rm $(docker ps -a -q)`
    - `sudo ls /var/lib/docker/containers/`
    - Tries, but conveniently fails to delete still-running containers

Ref: http://www.centurylinklabs.com/15-quick-docker-tips/

Proxy for Docker

Situation

I've setup a local caching proxy server, but not transparent proxy. For all those bunches of docker environments that I have, I want them all to use my local caching proxy server to get packages from, not each fetching out from the web on their own.

Conclusion

- set the http_proxy for the docker daemon in the /etc/default/docker file, if you pull docker images often.
- set the http_proxy environment variables with ENV in the Dockerfile, if you want to make use of cached packages during initial docker image building.
- set the http_proxy environment variables on the docker command line when invoking docker afterward:

```
$ docker run busybox env
HOME=/
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=1d9ba1029384

$ docker run -e HOME=/root -e http_proxy=http://my-proxy:3128/ busybox env
HOME=/root
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=5d42c805ef09
http_proxy=http://my-proxy:3128/
```

setup a http proxy for docker

https://groups.google.com/forum/#!topic/docker-user/gRXMTCLcRE0

set the http_proxy for the docker daemon, not the client process.

see the /etc/default/docker file for an example

Sven Dowideit

docker behind a proxy

You have wrong capitalization of environment variables in ENV. Correct one is `http_proxy`. Your example should be:

```
FROM ubuntu:13.10
ENV http_proxy <HTTP_PROXY>
ENV https_proxy <HTTPS_PROXY>
RUN apt-get update && apt-get upgrade
```

or

```
FROM centos
ENV http_proxy <HTTP_PROXY>
ENV https_proxy <HTTPS_PROXY>
RUN yum update
```

All variables specified in ENV are prepended to every RUN command. Every RUN command is executed in own container/environment, so it does not inherit variables from previous RUN commands!

Note: There is no need to call docker daemon with proxy for this to work, although if you want to pull images etc. you need to set the proxy for docker deamon too. You can set proxy for daemon in `/etc/default/docker` in Ubuntu (it does not affect containers setting).

Also, this can happen in case you **run your proxy on host** (i.e. localhost, 127.0.0.1). Localhost on host differ from localhost in container. In such case, you need to use another IP (like 172.17.42.1) to bind your proxy to or if you bind to 0.0.0.0, you can use 172.17.42.1 instead of 127.0.0.1 for connection from container during `docker build`.

You can also look for an example here: How to rebuild dockerfile quick by using cache?

answered Mar 4 2014

Jiri

Environment variables with Docker

http://serverascode.com/2014/05/29/environment-variables-with-docker.html

Docker supports setting environment variables with the -e switch.

Below is the simplest example. As can be seen, the FOO variable is indeed set to bar within the container.

```
# docker run -e FOO=bar busybox env

HOME=/

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

```
HOSTNAME=9c6f6cd077b2
FOO=bar
```

Multiple variables can be added with multiple -e switches.

```
# docker run -e FOO=bar -e BAR=foo busybox env
HOME=/
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=6cf2d6e8acb3
FOO=bar
BAR=foo
```

Using Docker to Build Firefox

http://gregoryszorc.com/blog/2013/05/19/using-docker-to-build-firefox/
https://gist.github.com/indygreg/5608534/raw/30704c59364ce7a8c69a02ee7f1cfb23d1ffcb2c/Dockerfile

Docker interests me because it allows simple environment isolation and repeatability. I can create a run-time environment once, package it up, then run it again on any other machine. Furthermore, everything that runs in that environment is isolated from the underlying host (much like a virtual machine). And best of all, everything is fast and simple.

To build Firefox with Docker, you'll first need to install Docker. That's pretty simple.

Then, it's just a matter of creating a new container with our build environment:

```
curl https://gist.github.com/indygreg/5608534/raw/30704c59364ce7a8c69a02ee7f1cfb23d1ffcb2c/Dockerfile | docker build
```

The output will look something like:

```
FROM ubuntu:12.10
MAINTAINER Gregory Szorc "gps@mozilla.com"
RUN apt-get update
===> d2f4faba3834
RUN dpkg-divert --local --rename --add /sbin/initctl && ln -s /bin/true /sbin/initctl
===> aff37cc837d8
RUN apt-get install -y autoconf2.13 build-essential unzip yasm zip
===> d0fc534feeee
RUN apt-get install -y libasound2-dev libcurl4-openssl-dev libdbus-1-dev libdbus-glib-1-dev libgtk2.0-dev libiw-dev libnotify-dev libxt-dev mesa-common-dev uuid-dev
===> 7c14cf7af304
```

```
RUN apt-get install -y binutils-gold
===> 772002841449
RUN apt-get install -y bash-completion curl emacs git man-db python-dev python-pip vim
===> 213b117b0ff2
RUN pip install mercurial
===> d3987051be44
RUN useradd -m firefox
===> ce05a44dc17e
Build finished. image id: ce05a44dc17e
ce05a44dc17e
```

As you can see, it is essentially *bootstrapping* an environment to build Firefox.

When this has completed, you can activate a shell in the container by taking the image id printed at the end and running it:

```
docker run -i -t ce05a44dc17e /bin/bash
# You should now be inside the container as root.
su - firefox
hg clone https://hg.mozilla.org/mozilla-central
cd mozilla-central
./mach build
```

If you want to package up this container for distribution, you just find its ID then export it to a tar archive:

```
docker ps -a
# Find ID of container you wish to export.
docker export 2f6e0edf64e8 > image.tar
# Distribute that file somewhere.
docker import - < image.tar
```

Simple, isn't it?

Docker Desktop: Your Desktop Over Ssh Running Inside Of A Docker Container

Motivation

I've worked for the last 4 years in the financial market and I'm a heavy Excel user. I also use the browser a lot and since I started my summer job as a financial intern at dotCloud I've been playing with docker. So, why don't we take advantage of the cloud and use these and other heavy applications (in terms of processing and memory usage) within a container (lighter than a normal VM).

Let's get started!

Docker Installation

- **On Linux:**
- Docker is available as a Ubuntu PPA (Personal Package Archive), hosted on launchpad which makes installing Docker on Ubuntu very easy.
  - `# Add the PPA sources to your apt sources list.`
  - `sudo apt-get install python-software-properties && sudo add-apt-repository ppa:dotcloud/lxc-docker`

  -
  - `# Update your sources`
  - `sudo apt-get update`
  -
  - `# Install, you will see another warning that the package cannot be authenticated. Confirm install.`
  - `sudo apt-get install lxc-docker`
- **On Mac:** Installation Tutorial
- **On Windows:** Installation Tutorial

Introduction

I used a Dockerfile in order to guarantee the integrity of the steps and also to automate the creation of the docker image. This Dockerfile creates a docker image that, once executed, generates a container that runs X11 and SSH services.

The ssh is used to forward X11 and provide you **encrypted data communication** between the docker container and your local machine.

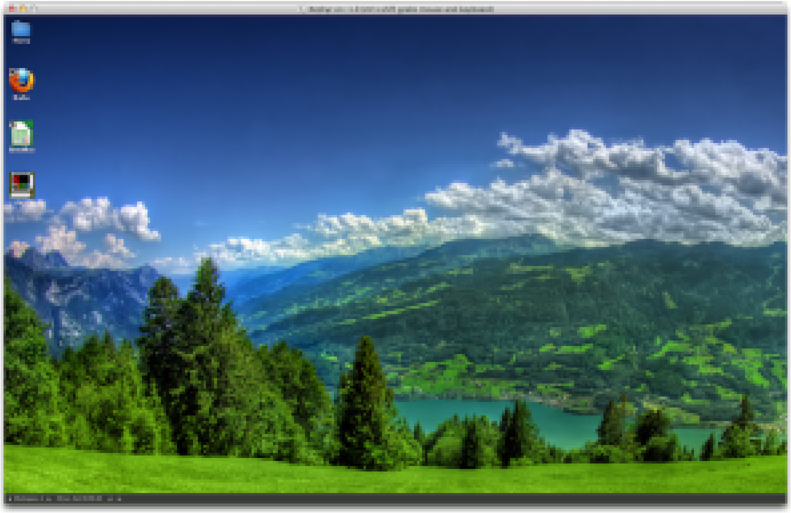
Xpra + Xephyr allows to display the applications running inside of the container such as Firefox, LibreOffice, xterm, etc. **with recovery session capabilities.** So, you can open your desktop anywhere without losing the status of your applications, even if the connection drops.

Xpra also uses a **custom protocol that is self-tuning and relatively latency-insensitive**, and thus is usable over worse links than standard X.

The applications can be rootless, so the **client machine manages the windows that are displayed**.

Fluxbox and ROX-Filer creates a very minimalist way to manage the windows and files.

creates a very minimalist way to manage files and icons on the desktop.



OBS: The client machine needs to have a X11 server installed (Xpra). See the "**Notes**" below.

Installation

Building the docker image

```
$ docker build -t [username]/docker-desktop
git://github.com/rogaha/docker-desktop.git
```

The parameter [-t] gives you the ability to name the image that is going to be created from the GitHub repository that contains the Dockerfile. In this case, the image will be called rogaha/docker-desktop. *This image has the docker-desktop (Xephyr+Fluxbox+ROX-filer), Firefox with Flash Player plugin, LibreOffice and xterm*. You can either use xterm or the ssh terminal to install/uninstall other GUI applications and use them remotely.

Verifying the existing images

```
$ docker images


REPOSITORY                                            TAG                 ID                  CREATED
       SIZE

base                                                  latest              b750fe79269d        3 months ago
24.65 kB (virtual 180.1 MB)

rogaha/docker-dektop                                  latest              5b07bd6027bd        5 days ago
12.29 kB (virtual  1.39 GB)

ubuntu                                                12.10               b750fe79269d        3 months ago
24.65 kB (virtual 180.1 MB)
```

As we can see the image rogaha/docker-desktop was created and is ready to run.

Running the docker image created -d: detached mode

```
$ CONTAINER_ID=$(docker run -d [username]/docker-desktop)
```

The environment variable $CONTAINER_ID contains the ID of the new running container created from the rogaha/docker-desktop image.

Getting the password generated during runtime

```
$ echo $(docker logs $CONTAINER_ID | sed -n 1p)

User: docker Password: xxxxxxxxxxxx
```

A new password is generated by PWGen every time that a container is created. The password contains 12 characters with at least one capital letter and one number.

Usage

Getting the container's external ssh port

```
$ docker port $CONTAINER_ID 22

49153 # This is the external port that forwards to the ssh service
```

```
        # running inside of the container as port 22
```

We are going to use this port later to connect to the machine where docker daemon is running.

Connecting to the container

Starting the a new session

First, we need to get the IP of the machine where docker is running.

```
$ ifconfig | grep "inet addr:"
inet addr:192.168.56.102 Bcast:192.168.56.255 Mask:255.255.255.0
# This is the LAN's IP for this machine
```

Then, we connect to that IP using the external port that we got previously (49153) in order to start a new session inside of the container

```
$ ssh docker@192.168.56.102 -p 49153 ./docker-desktop -s 800x600 -d 10
docker@192.168.56.102's password: xxxxxxxxxxxx

$ ./docker-desktop -h

-----------------------------------------------------------
Usage: docker-desktop [-s screen_size] [-d session_number]
-s : screen resolution (Default = 800x600
-d : session number (Default = 10)
-h : help
-----------------------------------------------------------
```

Attaching to the session started

```
$ xpra --ssh="ssh -p 49153" attach ssh:docker@192.168.56.102:10 # user@ip_address:session_number
docker@192.168.56.102's password: xxxxxxxxxxxx
```

If you want to execute rootless applications, you just need to connect to the container via ssh and type: DISPLAY=:[session_number] [program_name] &

Eg. DISPLAY=:10 firefox &

Conclusion

It took me only a few hours to create this Dockerfile and now I'm able to use it anywhere at any time and everything is configured and ready to run.

Developers can save a lot of time with docker by automating their workflow and normal users can take advantage of the cloud and run heavy applications remotely and save their local processing and memory usage.

If you have any questions, please feel free to contact me at roberto.hashioka@dotcloud.com

Other references

Create Lightweight Docker Containers With Buildroot

http://blog.docker.io/2013/06/create-light-weight-docker-containers-buildroot/

Building Your Own Platform Service Using Docker

http://vimeo.com/67284401

How to use official Debian images from Docker

```
$ docker pull debian:sid
Pulling repository debian
77e97a48ce6a: Download complete
...
08ef8d4565f6: Download complete
```

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
debian              sid                 77e97a48ce6a        30 hours ago        114.9 MB
```

Ref, https://registry.hub.docker.com/_/debian/ for the list of tags. They are built with Stackbrew, a web-application that performs *continuous building* of the docker standard library.

How to build base boxes for Docker

Docker requires that all personal builds to be under a username. You don’t have to register it in the docker registry to create your own images. However, if you do, you can also upload your own images in this repository. Note that you don’t need to upload your own images for you to use them yourself, but only for others to use instead.

The following demonstration use `sfxpt` as the docker username.

Using mkimage.sh

The mkimage.sh from docker contrib script may be used, alongside the companion script mkimage/debootstrap to create a Docker image for Debian (or Ubuntu) using debootstrap.

NB, under Ubuntu, the mkimage.sh is not pre-installed, so we have to install it ourselves:

```
 sudo sh -xc 'curl https://raw.githubusercontent.com/dotcloud/docker/master/contrib/mkimage.sh >
/usr/local/sbin/mkimage.sh'
 sudo chmod 755 /usr/local/sbin/mkimage.sh
 sudo mkdir /usr/local/sbin/mkimage
 sudo sh -xc 'curl https://raw.githubusercontent.com/dotcloud/docker/master/contrib/mkimage/debootstrap >
/usr/local/sbin/mkimage/debootstrap'
 sudo chmod 755 /usr/local/sbin/mkimage/debootstrap
 mkimage.sh
```

You may use it to create an image for the stable suite (using the minbase debootstrap variant) with :

```
sudo mkimage.sh -t sfxpt/debian debootstrap --variant=minbase stable
```

The image may then be used with :

```
docker run -i sfxpt/debian echo "hello world"
```

Contrary to the former mkimage-debootstrap.sh (now deprecated, see below), contrib/mkimage/debootstrap does not need the mandatory variant parameter, and can invoke debootstrap without a specific variant, using its default settings.

Sources::
https://wiki.debian.org/Cloud/CreateDockerImage
http://my.oschina.net/guol/blog/264376?p=%7B%7BcurrentPage+1%7D%7D

Here is an actual example:

```
$ sudo mkimage.sh -t sfxpt/debian debootstrap --variant=minbase wheezy
+ mkdir -p /var/tmp/docker-mkimage.7V1p40bOA8/rootfs
+ debootstrap --variant=minbase wheezy /var/tmp/docker-mkimage.7V1p40bOA8/rootfs
I: Retrieving Release
I: Retrieving Packages
I: Validating Packages
…
I: Base system installed successfully.
+ echo exit 101 > '/var/tmp/docker-mkimage.7V1p40bOA8/rootfs/usr/sbin/policy-rc.d'
+ chroot /var/tmp/docker-mkimage.7V1p40bOA8/rootfs dpkg-divert --local --rename --add /sbin/initctl
…
+ chroot /var/tmp/docker-mkimage.7V1p40bOA8/rootfs bash -c 'apt-get update && apt-get dist-upgrade -y'
Get:1 http://ftp.us.debian.org wheezy Release.gpg [1655 B]
```

```
…
Get:3 http://security.debian.org/ wheezy/updates/main multiarch-support amd64 2.13-38+deb7u4 [151 kB]
Fetched 5643 kB in 7s (752 kB/s)
debconf: delaying package configuration, since apt-utils is not installed
Can not write log, openpty() failed (/dev/pts not mounted?)
(Reading database ... 6609 files and directories currently installed.)
Preparing to replace libc-bin 2.13-38+deb7u2 (using .../libc-bin_2.13-38+deb7u4_amd64.deb) ...
Unpacking replacement libc-bin ...
Can not write log, openpty() failed (/dev/pts not mounted?)
Setting up libc-bin (2.13-38+deb7u4) ...
Can not write log, openpty() failed (/dev/pts not mounted?)
(Reading database ... 6608 files and directories currently installed.)
Preparing to replace libc6:amd64 2.13-38+deb7u2 (using .../libc6_2.13-38+deb7u4_amd64.deb) ...
…
+ rm -rf /var/tmp/docker-mkimage.7V1p40bOA8/rootfs
+ docker build -t sfxpt/debian /var/tmp/docker-mkimage.7V1p40bOA8
Sending build context to Docker daemon 21.19 MB
Sending build context to Docker daemon
Step 0 : FROM scratch
Pulling repository scratch
511136ea3c5a: Download complete  ---> 511136ea3c5a
Step 1 : ADD rootfs.tar.xz /
 ---> bf5fd0f9cf99
Removing intermediate container da57ecb27894
Step 2 : CMD ["/bin/bash"]
 ---> Running in 9a72f2a136ce
 ---> a8390a40ca02
Removing intermediate container 9a72f2a136ce
Successfully built a8390a40ca02
+ rm -rf /var/tmp/docker-mkimage.7V1p40bOA8
```

```
$ docker run -i sfxpt/debian echo "hello world"
hello world
```

```
$ docker images
REPOSITORY       TAG             IMAGE ID         CREATED            VIRTUAL SIZE
sfxpt/debian     latest          a8390a40ca02     6 minutes ago      83.97 MB
```

Ah, here, tagging wheezy as “latest” should be a bug. Filed a bug report at https://github.com/docker/docker/issues/7852. The remedy is:

```
docker tag sfxpt/debian sfxpt/debian:wheezy
```

Update, to prevent it and/or all kinds of ubuntu releases (saucy, trusty, etc) being tagged as the `latest`, specify the tag as "my/ubuntu:saucy", for example.

PS, one key `mkimage-debootstrap.sh` feature missing from `mkimage.sh` for the moment is the capability of specifying proxy servers. It is vitally important for those people who are behind some proxy servers, or for those people that are using caching servers to cache Debian packages instead of downloading them over and over again when building each docker image. Details at https://github.com/docker/docker/issues/7853.

Using mkimage-debootstrap.sh

NB, this method is now deprecated, and ~~not working any more~~.

To make debian-based boxes, use the mkimage-debootstrap.sh in contrib. It is better than mkimage-debian.sh script in that, “it includes all sorts of goodies, including a bunch of command line arguments, very much improved Ubuntu support, and adding some nice repositories to `/etc/apt/sources.list` by default (especially security-related repositories).

Here's a list of a few of the things:

- options to control certain script parameters
- an option specifically for creating tarballs directly, esp. for dockerbrew
- the addition of the updates and security repositories for Debian images
- the addition of the universe, updates, and security repositories for Ubuntu images
- more correct tagging of Debian images
- tagging of Ubuntu image versions (12.04, 12.10, etc) and latest tag for LTS”

First make sure the debootstrap is installed:

```
 sudo apt-get install debootstrap
```

Then get the latest mkimage-debootstrap.sh:

```
 sudo sh -xc 'curl https://raw.githubusercontent.com/dotcloud/docker/master/contrib/mkimage-debootstrap.sh >
/usr/sbin/mkimage-debootstrap.sh'
 sudo chmod 755 /usr/sbin/mkimage-debootstrap.sh
```

The build step is very simple:

```
$ mkimage-debootstrap.sh sfxpt/debian sid
+ mkdir -p /tmp/docker-rootfs-debootstrap-sid-25468-3819
+ sudo http_proxy= debootstrap --verbose --variant=minbase --include=iproute,iputils-ping --arch=amd64 sid
/tmp/docker-rootfs-debootstrap-sid-25468-3819 ''
I: Retrieving Release
W: Cannot check Release signature; keyring file not available /usr/share/keyrings/debian-archive-keyring.gpg
I: Retrieving Packages
I: Validating Packages
```

```
I: Resolving dependencies of required packages...
I: Resolving dependencies of base packages...
I: Found additional required dependencies: insserv libaudit-common libaudit1 libbz2-1.0 libcap2 libdb5.1 libsemanage-common libsemanage1 libslang2 libustr-1.0-1
I: Found additional base dependencies: debian-archive-keyring gnupg gpgv iproute2 libapt-pkg4.12 libffi6 libgcrypt11 libgnutls-openssl27 libgnutls26 libgpg-error0 libp11-kit0 libreadline6 libstdc++6 libtasn1-3 libusb-0.1-4 readline-common
I: Checking component main on http://ftp.us.debian.org/debian...
I: Retrieving libacl1 2.2.52-1
I: Validating libacl1 2.2.52-1
I: Retrieving apt 0.9.12.1
I: Validating apt 0.9.12.1
. . .
+ sudo tar --numeric-owner -c .
+ sudo docker import - sfxpt/debian sid
b02adec6141f
+ sudo docker run -i -t sfxpt/debian:sid echo success
success
+ '[' -z '' ']'
+ case "$lsbDist" in
+ '[' sid = wheezy -o sid = stable ']'
+ cd /usr/sbin
+ sudo rm -rf /tmp/docker-rootfs-debootstrap-sid-25468-3819
```

As we can see the image sfxpt/debian:sid was created and runs fine.

Note,

- It’s better not to specific the repo url, because it’ll use the default debootstrap mirror if one is not provided, which will almost always be a better choice.
- And the end of the script, it use “sudo docker run” to verify that the image was created and runs fine.

To build an Ubuntu docker image:

```
$ mkimage-debootstrap.sh sfxpt/ubuntu saucy
. . .
I: Checking component main on http://archive.ubuntu.com/ubuntu...
I: Retrieving apt 0.9.9.1~ubuntu3
I: Validating apt 0.9.9.1~ubuntu3
. . .
+ sudo docker tag sfxpt/ubuntu:saucy sfxpt/ubuntu 13.10
```

That’s it.

自制docker镜像

http://my.oschina.net/guol/blog/264376?p=%7B%7BcurrentPage+1%7D%7D

发表于 2014-05-13

上一篇我简单的操作了几个docker命令，但是有时候使用公共仓库下载的镜像，总是和业务有些不太完全符合，因此我要定制符合自己业务情况的基础镜像，自制镜像的方法多种多样，目前介绍三种方法：

1 利用已有的ISO镜像制作基本镜像，以ubuntu14.04为例

?

```
mount -o loop ubuntu-14.04-server-amd64.iso /media/cdrom
tar -C /media/cdrom -c . | docker import - guol/ubuntu14
```

这样很块就可以制作好一个基础镜像，但是这个镜像制作出来很大，里面包含了一些我们不需要的软件，如果你想裁剪的话，可以使用chroot命令进入基础镜像里面，做一些软件包的删减。

2 debian/ubuntu系列有个工具debootstrap，可以用来制作较小的镜像，但是也有300M，可以修改构建脚本，进行裁剪，以ubuntu14.04为例

?

```
debootstrap --variant=buildd --arch=amd64 trusty trusty http://mirrors.163.com/ubuntu/
tar -C trusty -c . | docker import - trusty
```

注意：debootstrap的使用方法

debootstrap [OPTION...] SUITE TARGET [MIRROR [SCRIPT]]

SUITE 是一些linux发行版的名称，默认在/usr/share/debootstrap/scripts/目录下有的linux发行版构建脚本才能使用，默认debootstrap就是根据这些脚本做构建的

MIRROR 一般默认使用官方的源构建，这样比较慢，一般都换成国内源，这个源默认是在/usr/share/debootstrap/scripts/的构建脚本中设置的，直接加在命令行会覆盖里面的设置

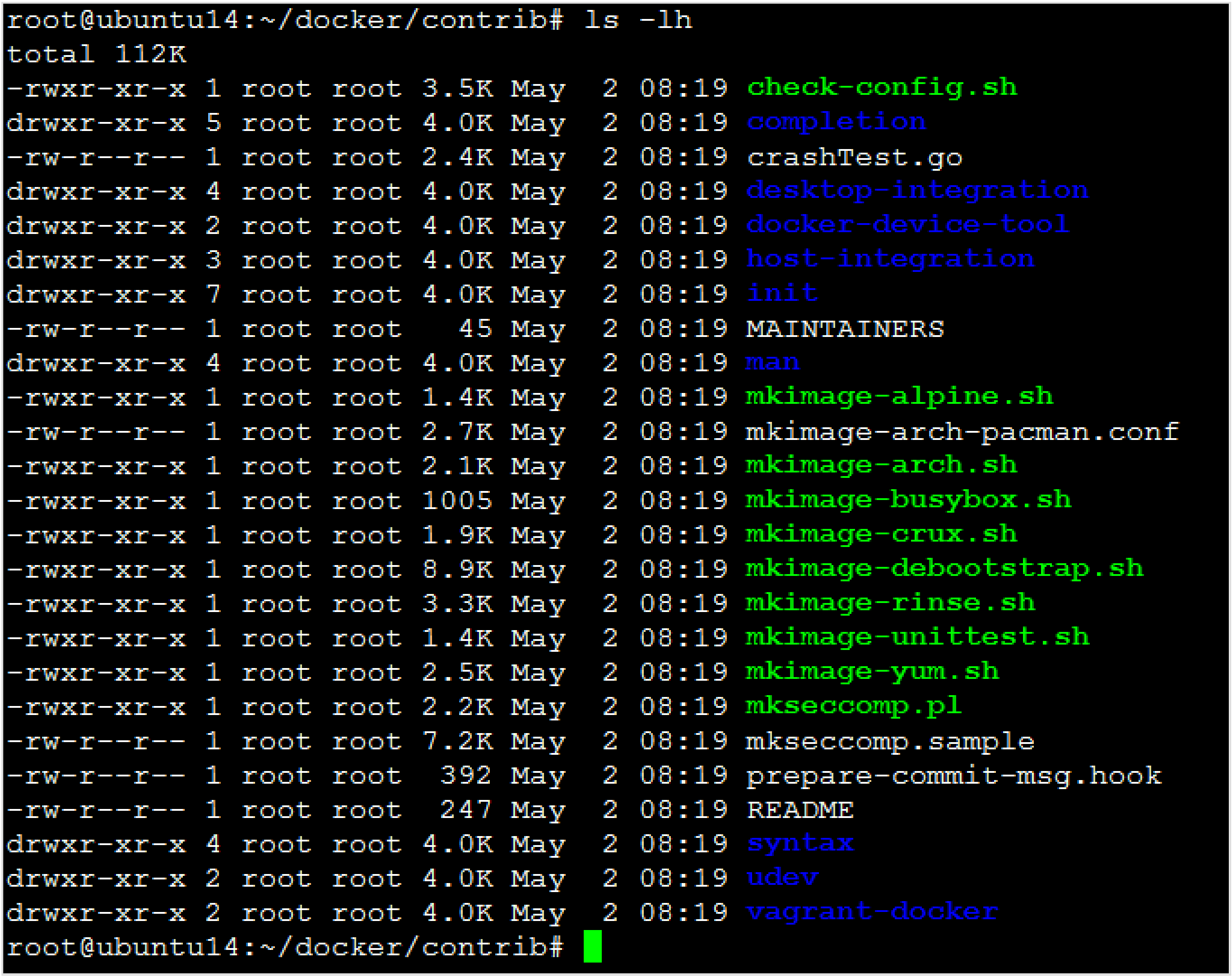
SCRIPT debootstrap根据什么脚本做镜像构建，默认使用/usr/share/debootstrap/scripts/目录下的，你也可以自己编写。

3 使用官方的构建脚本，在github上官方源码里面就有。

?

```
git clone https://github.com/dotcloud/docker.git
cd docker/contrib
```

来看看下载的源码里都有什么，进入contrib目录

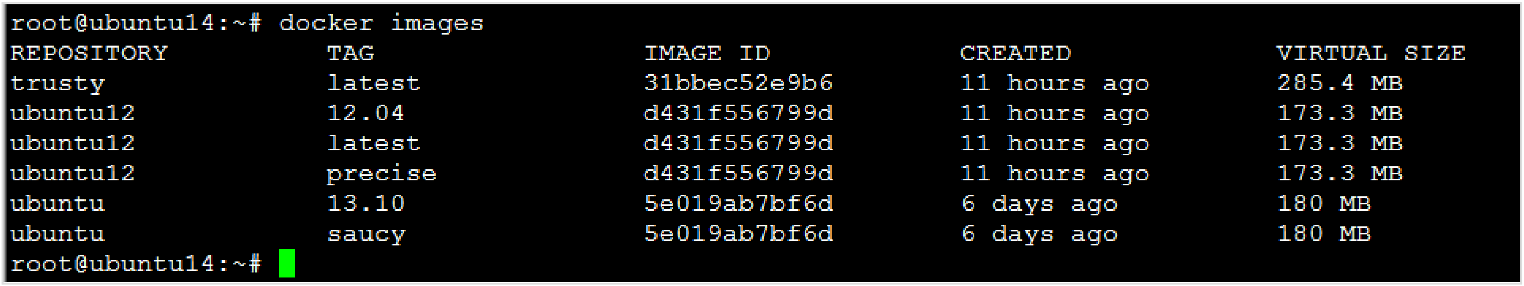


```
root@ubuntu14:~/docker/contrib# ls -lh
total 112K
-rwxr-xr-x 1 root root 3.5K May  2 08:19 check-config.sh
drwxr-xr-x 5 root root 4.0K May  2 08:19 completion
-rw-r--r-- 1 root root 2.4K May  2 08:19 crashTest.go
drwxr-xr-x 4 root root 4.0K May  2 08:19 desktop-integration
drwxr-xr-x 2 root root 4.0K May  2 08:19 docker-device-tool
drwxr-xr-x 3 root root 4.0K May  2 08:19 host-integration
drwxr-xr-x 7 root root 4.0K May  2 08:19 init
-rw-r--r-- 1 root root   45 May  2 08:19 MAINTAINERS
drwxr-xr-x 4 root root 4.0K May  2 08:19 man
-rwxr-xr-x 1 root root 1.4K May  2 08:19 mkimage-alpine.sh
-rw-r--r-- 1 root root 2.7K May  2 08:19 mkimage-arch-pacman.conf
-rwxr-xr-x 1 root root 2.1K May  2 08:19 mkimage-arch.sh
-rwxr-xr-x 1 root root 1005 May  2 08:19 mkimage-busybox.sh
-rwxr-xr-x 1 root root 1.9K May  2 08:19 mkimage-crux.sh
-rwxr-xr-x 1 root root 8.9K May  2 08:19 mkimage-debootstrap.sh
-rwxr-xr-x 1 root root 3.3K May  2 08:19 mkimage-rinse.sh
-rwxr-xr-x 1 root root 1.4K May  2 08:19 mkimage-unittest.sh
-rwxr-xr-x 1 root root 2.5K May  2 08:19 mkimage-yum.sh
-rwxr-xr-x 1 root root 2.2K May  2 08:19 mkseccomp.pl
-rw-r--r-- 1 root root 7.2K May  2 08:19 mkseccomp.sample
-rw-r--r-- 1 root root  392 May  2 08:19 prepare-commit-msg.hook
-rw-r--r-- 1 root root  247 May  2 08:19 README
drwxr-xr-x 4 root root 4.0K May  2 08:19 syntax
drwxr-xr-x 2 root root 4.0K May  2 08:19 udev
drwxr-xr-x 2 root root 4.0K May  2 08:19 vagrant-docker
root@ubuntu14:~/docker/contrib# 
```

可以看到很多镜像构建脚本，根据你的操作系统选择脚本，我使用的是mkimage-debootstrap.sh，不过有个问题就是采用自带脚本安装时该脚本只支持debian6 ubuntu12，因为脚本中只指定了这两个发行版，不过可以自己手动修改脚本，以支持更多的发行版。该脚本会在/tmp目录下创建一个临时目录来构建镜像，构建成功后自动import，然后自动删除该构建目录，这个制作出来的镜像比较小180M左右。

?

```
./mkimage-debootstrap.sh ubuntu12 precise http://mirrors.sohu.com/ubuntu/
```



```
root@ubuntu14:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
trusty              latest              31bbec52e9b6        11 hours ago        285.4 MB
ubuntu12            12.04               d431f556799d        11 hours ago        173.3 MB
ubuntu12            latest              d431f556799d        11 hours ago        173.3 MB
ubuntu12            precise             d431f556799d        11 hours ago        173.3 MB
ubuntu              13.10               5e019ab7bf6d        6 days ago          180 MB
ubuntu              saucy               5e019ab7bf6d        6 days ago          180 MB
root@ubuntu14:~# 
```

可以看到自动制作镜像，并且自动提交。

5-22

docker更新真是快啊，今天git下载了最新的源码，发现mkimage-debootstrap.sh已经支持ubuntu14.04了，不过该脚本以后要放弃使用了，官方建议使用同一目录下的mkimage.sh脚本来构建镜像。

例如mkimage.sh -t someuser/debian debootstrap --variant=minbase jessie -t 就是你的tag设置 这个应该容易明白 debootstrap 就是构建镜像的脚本，debian/ubuntu在构建镜像时都是使用debootstrap这个软件包构建的，你把这个软件安装完毕后，你在ubuntu下进入/usr/share/debootstrap/scripts/这个目录就会看到很多构建镜像的脚本。

搭建私有仓库

有时候从公有仓库获取镜像的时候，出现网络问题，经常就不通了，很无奈，所以还是搭建一个自己的私有仓库比较靠谱。关于搭建私有仓库的办法，官方已经给出方案了，那就是docker-registry。例如：

. . .

How to build and publish base boxes for Docker

http://crohr.me/journal/2013/docker-repository-create-base-boxes.html

Docker comes with a registry where you can find images to use for your containers. You can also create and upload your own images in this repository, which is the subject of this article.

- Create an account on the Docker index website: https://index.docker.io.
- On your box, login:

-
  ```
  $> docker login
  Username (): crohr
  Password:
  Email (): hi@example.com
  Login Succeeded
  ```
- For debian-based boxes, just use the mkimage-debian.sh script in contrib/:

-
  ```
  $> ./mkimage-debian.sh crohr/ubuntu lucid http://archive.ubuntu.com/ubuntu
  ...
  + img=f7e10b978e8c
  + docker tag f7e10b978e8c crohr/ubuntu lucid
  + '[' lucid = wheezy ']'
  + docker run -i -t crohr/ubuntu:lucid echo success
  success
  + '[' lucid '!=' sid -a lucid '!=' unstable ']'
  ++ docker run crohr/ubuntu:lucid cat /etc/debian_version
  + ver=squeeze/sid
  + docker tag f7e10b978e8c crohr/ubuntu squeeze/sid
  2013/05/24 17:05:40 error: Illegal tag name: squeeze/sid
  ```
- Note: The second tag is not created, but it does not matter (?).
- Then, check that your image is indeed created:

-
  ```
  $> docker images
  REPOSITORY          TAG                 ID                  CREATED
  crohr/ubuntu        lucid               f7e10b978e8c        34 minutes ago
  ubuntu              latest              8dbd9e392a96        6 weeks ago
  ubuntu              12.10               b750fe79269d        8 weeks ago
  ubuntu              12.04               8dbd9e392a96        6 weeks ago
  ubuntu              quantal             b750fe79269d        8 weeks ago
  ubuntu              precise             8dbd9e392a96        6 weeks ago
  ```

- Now push it to the central repository:

-
  ```
  $> docker push crohr/ubuntu
   The push refers to a repository [crohr/ubuntu] (len: 1)
   Processing checksums
   Sending image list
   Pushing repository crohr/ubuntu to registry-1.docker.io (1 tags)
   Pushing f7e10b978e8c15bde845374c490862befb181f0943a0ae2076d070573a435b16
   145858560/145858560 (100%)
   Pushing tags for rev [f7e10b978e8c15bde845374c490862befb181f0943a0ae2076d070573a435b16] on
  {registry-1.docker.io/users/crohr/ubuntu/lucid}
  ```

That’s it! Now other people can pull your images using `docker pull crohr/ubuntu`.

Self-hosted docker registry

http://blog.docker.io/2013/07/docker-0-5-0-external-volumes-advanced-networking-self-hosted-registry/#self_hosted_registry

This release includes major improvements to the self-hosted registry. It is now much easier to store and distribute docker images privately within your organization. You can also combine multiple registries to reflect your organization: for example, you might store your open-source images on the public registry, company-wide base images on a company-wide registry, application nightly builds in a team-specific registry, and so on.

Self-hosted registries are completely independent of the public registry: images stored on it are completely private and not publicly searchable or accessible in any way. You can control access to your registry using standard HTTP proxying and authentication.

To find out which registry an image comes from, simply look at its name: the URL of the registry is used as a prefix. For example, `images.myorganization.com/ubuntu` refers to an image in the `images.myorganization.com` registry. Image names with no URL, like `ubuntu` or `shykes/couchdb` always refer to the public registry.

To pull an image from a registry, simply include the registry url in the image name:

```
docker pull myregistry.mydomain.com/myapp
```

Conversely, to push an image to a registry, you must first tag it with the appropriate name:

```
# Tag to create a repository with the full registry location.

# The location (e.g. myregistry.mydomain.com) becomes
```

```
# a permanent part of the repository name
docker tag 0u812deadbeef myregistry.mydomain.com/myapp

# Push the new repository to its home location
docker push myregistry.mydomain.com/myapp
```

And that’s it! No other configuration necessary.

Flatten a Docker container or image

http://tuhrig.de/flatten-a-docker-container-or-image/

Thomas Uhrig in Allgemein | 31/03/2014

Docker containers and respectively images can become fairly large. I recently worked with a Docker image which was over 7 GB big. However, it is pretty easy to flatten an image at the end.

Difference between save and export

As I described in my last post (http://tuhrig.de/difference-between-save-and-export-in-docker), there are **two ways to persist a Docker images or container**:

- A Docker *image* can be *saved* to a tarball and *loaded* back again. This will preserve the history of the image.

  ```
  # save the image to a tarball
  docker save <IMAGE NAME> > /home/save.tar

  # load it back
  docker load < /home/save.tar
  ```

- A Docker *container* can be *exported* to a tarball and *imported* back again. This will *not* preserve the history of the container.

  ```
  # export the container to a tarball
  docker export <CONTAINER ID> > /home/export.tar

  # import it back
  cat /home/export.tar | docker import - some-name:latest
  ```

No history

We can use this mechanism to flatten and shrink a Docker container. If we save an image to the disk, its whole history will be preserved, but if we export a container, its history gets lost and the resulting tarball will be much smaller.

We can see the history of a image be running `docker tag <LAYER ID> <IMAGE NAMEgt;`:

```
vagrant@ubuntu-13:~$ sudo docker images --tree
├─f502877df6a1 Virtual Size: 2.489 MB Tags: busybox-1-export:latest
└─511136ea3c5a Virtual Size: 0 B
  └─bf747efa0e2f Virtual Size: 0 B
    └─48e5f45168b9 Virtual Size: 2.489 MB
      └─769b9341d937 Virtual Size: 2.489 MB
        └─227516d93162 Virtual Size: 2.489 MB Tags: busybox-1:latest
```

So if we export a container (either an already running one or just start a new one from an image) it will lose its history and all previous layers. This will make it impossible to make a rollback to a certain layer, but it will also shrink the image. My >7 GB image is now >3 GB large, which saves more than 50% of disk space.

Flatten a Docker container

So it is only possible to “flatten” a Docker container, not an image. So we need to start a container from an image first. Then we can export and import the container in one line:

```
docker export <CONTAINER ID> | docker import - some-image-name:latest
```

What else?

You can use some common Linux tricks to shrink Docker images. One simple trick is to clear the cache of the package manager. So depending on which base image you use you can do something like this (for an Ubuntu/Debian system, for more see here):

```
# clean apt cache
apt-get clean
```

Resources

- http://stackoverflow.com/questions/22713551/how-to-flatten-a-docker-image

- http://docs.docker.io/en/latest/reference/commandline/cli
- http://unix.stackexchange.com/questions/75932/what-stuff-can-be-safely-removed-for-disk-space-sake
-

Copy file between container and host

Docker - copy file from container to host

http://stackoverflow.com/questions/22049212/docker-copy-file-from-container-to-host

n order to copy a file from a container to the host you can use

`docker cp <containerId>:/file/path/within/container /host/path/target`.

Feb 26 at 18:31

creack

Copying files from host to docker container

http://stackoverflow.com/questions/22907231/copying-files-from-host-to-docker-container

> *How can I copy files from the host to the container?*

1. Get container name or short container id :
   docker ps
2. Get full container id
   docker inspect -f '{{.Id}}' SHORT_CONTAINER_ID-or-CONTAINER_NAME
3. copy file :
   sudo cp path-file-host /var/lib/docker/aufs/mnt/FULL_CONTAINER_ID/PATH-NEW-FILE

EXAMPLE :

$docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS **NAMES**

d8e703d7e303 solidleon/ssh:latest /usr/sbin/sshd -D **cranky_pare**

$docker inspect -f '{{.Id}}' cranky_pare

or

$docker inspect -f '{{.Id}}' d8e703d7e303

d8e703d7e3039a6df6d01bd7fb58d1882e592a85059eb16c4b83cf91847f88e5

$sudo cp file.txt /var/lib/docker/aufs/mnt/**d8e703d7e3039a6df6d01bd7fb58d1882e592a85059eb16c4b83cf91847f88e5**/root/file.txt

Jul 17

user2357585

ugly but works reliably:

```
docker run -i ubuntu /bin/bash -c 'cat > file' < file
```

Jun 11

Erik

Where are Docker images stored?

http://blog.thoward37.me/articles/where-are-docker-images-stored/

***Summary**, all docker related files are stored under the folder /var/lib/docker. So to make all my docker files available to my different multi-boot Linux OSs, I just symlink /var/lib/docker to a central place which is available to all my multi-boot OSs.*

by troy howard

If you're just starting out with Docker, it's super easy to follow the examples, get started and run a few things. However, moving to the next step, making your own Dockerfiles, can be a bit confusing. One of the more common points of confusion seems to be:

> *Where are my Docker images stored?*

I know this certainly left me scratching my head a bit. Even worse, as a n00b, the last thing you want to do is publish your tinkering on the public Docker Index.

So let me start with this small assurance; *Nothing you do will become public* especially if:

1. You haven't made an account on the public index.
2. You haven't run `docker login` to authenticate via the command-line client.
3. You don't run `docker push`, to push an image up to the index.

Vocabulary

One of the things that contributes to much of the confusion around Docker is the language that's used. There's a lot of terminology which seem to overlap, or is a bit ambiguous, used somewhat incorrectly, or has a well-established meaning that is different from how Docker uses it.

I'll try to clear those up here, in a quick vocabulary lesson.

Image vs Dockerfile

This one is the least confusing, but it's an important distinction. Docker uses *images* to run your code, not the *Dockerfile*. The *Dockerfile* is used to build the image when you run `docker build`.

If you go browsing around on the Docker Index, you'll see lots of images listed there, but weirdly, you can't see the *Dockerfile* that built them. The image is an *opaque asset* that is compiled from the *Dockerfile*.

When you run `docker push` to publish an image, it's not publishing your source code, it's publishing the image that was built from your source code.

Registry vs Index

The next weird thing is the idea of a *Registry* and an *Index*, and how these are separate things.

An *index* manages user accounts, permissions, search, tagging, and all that nice stuff that's in the public web interface.

A *registry* stores and serves up the actual image assets, and it delegates authentication to the *index*.

When you run `docker search`, it's searching the *index*, not the *registry*. In fact, it might be searching *multiple registries* that the index is aware of.

When you run `docker push` or `docker pull`, the *index* determines if you are allowed to access or modify the image, but the *registry* is the piece that stores it or sends it down the wire to you after the *index* approves the operation. Also, the *index* figures out which *registry* that particular image lives in and forwards the request appropriately.

Beyond that, when you're working locally and running commands like `docker images`, you're interacting with something that is neither an index or a registry, but a little of both.

Repository

Docker's use of this word is similar to its use at Github, and other source control systems, but also, kind of not.

Three common head-scratching questions are:

- What's the difference between a repository and a registry?
- What's the difference between a repository and an image?
- What's the difference between a repository and an index username?

In fact, this is a problem, because a *repository* is all of those things and not really any of them either. Further, when you run `docker images` you get output like this:

```
$ docker images
REPOSITORY       TAG              IMAGE ID          CREATED           SIZE
ubuntu           12.04            8dbd9e392a96      8 months ago      131.3 MB (virtual 131.3 MB)
ubuntu           latest           8dbd9e392a96      8 months ago      131.3 MB (virtual 131.3 MB)
ubuntu           precise          8dbd9e392a96      8 months ago      131.3 MB (virtual 131.3 MB)
ubuntu           12.10            b750fe79269d      8 months ago      24.65 kB (virtual 179.7 MB)
ubuntu           quantal          b750fe79269d      8 months ago      24.65 kB (virtual 179.7 MB)
```

So, the list of *images* seems to be a list of repositories? Huh? Actually the images are the GUIDs, but that's not how you interact with them.

Let's start over with this.

When you run `docker build` or `docker commit`, you can specify a name for the *image*. The name is usually in the format of `username/image_name`, but it doesn't have to be. It could be anything, and it could even be the same as something well known and published.

However, when the time comes to `docker push`, the *index* will look at the name, and will check to see if it has a matching *repository*. If it does, it will check to see if you have access to that *repository*, and if so, allow you to push the new version of the *image* to it. So, a *registry* holds a collection of named *repositories*, which themselves are a *collection of images* tracked by GUIDs. This is also where *tags* come in. You can tag an image, and store multiple versions of that image with different GUIDs in a single named *repository*, access different tagged versions of an image with a special syntax like `username/image_name:tag`.

```
$ docker images
REPOSITORY        TAG              IMAGE ID          CREATED           SIZE
ubuntu            12.04            8dbd9e392a96      8 months ago      131.3 MB (virtual 131.3 MB)
ubuntu            latest           8dbd9e392a96      8 months ago      131.3 MB (virtual 131.3 MB)
ubuntu            precise          8dbd9e392a96      8 months ago      131.3 MB (virtual 131.3 MB)
ubuntu            12.10            b750fe79269d      8 months ago      24.65 kB (virtual 179.7 MB)
ubuntu            quantal          b750fe79269d      8 months ago      24.65 kB (virtual 179.7 MB)
```

If we look at the output from `docker images` again, now it makes a little more sense. We have five different versions of the image named `ubuntu`, each one tagged slightly differently. The repository holds all of those under that name `ubuntu`. So, while it may seem like `ubuntu` is an *image name*, it's actually a *repository name*, indicating where it came from, or where it should go during a push.

Further, the repository name has a specific schema to it. An *index* can parse out the username from first part, and figure out where it is.

So, this the confusing part: Suppose there's a Docker image called `thoward/scooby_snacks`.

The official "repository name" is thoward/scooby_snacks, even though we would normally think of the repository as just being scooby_snacks (eg, in GitHub, or elsewhere).

In fact, when the Docker documentation refers to a *repository*, it sometimes means the whole thing, username included, and sometimes only means the part after the username.

That's because some repositories don't have usernames (like ubuntu). The username is very important to handle separately, because it's used for authentication by the *index*, so that part of the repository name has its own semantics separate from the name, when it's there.

Local Storage on the Docker Host

So far I've been explaining the intricacies of remote storage, and how that relates to the confusing vocabulary, but running docker images shows you only what is local to your machine.

Where is this stuff? The first place to look is in /var/lib/docker/.

Open up the file repositories to find a JSON list of the repositories on your host:

```
$ sudo cat /var/lib/docker/repositories | python -mjson.tool
{
    "Repositories": {
        "ubuntu": {
            "12.04": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
            "12.10": "b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc",
            "latest": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
            "precise": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
            "quantal": "b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc"
        }
    }
}
```

Hey, that matches the output from docker images!

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              12.04               8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              latest              8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              precise             8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              12.10               b750fe79269d        8 months ago        24.65 kB (virtual 179.7 MB)
ubuntu              quantal             b750fe79269d        8 months ago        24.65 kB (virtual 179.7 MB)
```

Checkout what's in /var/lib/docker/graph/:

```
$ sudo ls -al /var/lib/docker/graph
total 24
drwx------ 6 root root 4096 Nov 22 06:52 .
drwx------ 5 root root 4096 Dec 13 04:25 ..
drwxr-xr-x 3 root root 4096 Dec 13 04:26 27cf784147099545
drwxr-xr-x 3 root root 4096 Nov 22 06:52 8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c
drwxr-xr-x 3 root root 4096 Nov 22 06:52 b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc
drwx------ 3 root root 4096 Nov 22 06:52 _tmp
```

Not terribly friendly, but we can see how Docker is keeping track of these, based on the repositories JSON file which holds a mapping of repository names and tags, to the underlying image GUIDs.

We have two images from the ubuntu repository, with the tags **12.04**,**precise**, and **latest** all corresponding to the image with id8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c (or8dbd9e392a96 for short).

So what's actually stored there?

```
$ sudo ls -al /var/lib/docker/graph/8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c
total 20
drwxr-xr-x  3 root root 4096 Nov 22 06:52 .
drwx------  6 root root 4096 Nov 22 06:52 ..
-rw-------  1 root root  437 Nov 22 06:51 json
drwxr-xr-x 22 root root 4096 Apr 11  2013 layer
-rw-------  1 root root    9 Nov 22 06:52 layersize
```

The entries here are:

- json - holds metadata about the image
- layersize - just a number, indicating the size of the layer
- layer/ - sub-directory that holds the rootfs for the container image

```
$ sudo cat /var/lib/docker/graph/8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c/json | python
-mjson.tool
{
    "comment": "Imported from -",
    "container_config": {
        "AttachStderr": false,
        "AttachStdin": false,
        "AttachStdout": false,
        "Cmd": null,
        "Env": null,
        "Hostname": "",
        "Image": "",
        "Memory": 0,
        "MemorySwap": 0,
        "OpenStdin": false,
        "PortSpecs": null,
        "StdinOnce": false,
        "Tty": false,
        "User": ""
    },
    "created": "2013-04-11T14:13:15.57812-07:00",
    "docker_version": "0.1.4",
    "id": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c"
}
```

```
$ sudo cat /var/lib/docker/graph/8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c/layersize
131301903
```

```
$ sudo ls -al /var/lib/docker/graph/8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c/layer
total 88
drwxr-xr-x 22 root root 4096 Apr 11  2013 .
drwxr-xr-x  3 root root 4096 Nov 22 06:52 ..
drwxr-xr-x  2 root root 4096 Apr 11  2013 bin
drwxr-xr-x  2 root root 4096 Apr 19  2012 boot
drwxr-xr-x  4 root root 4096 Nov 22 06:51 dev
drwxr-xr-x 41 root root 4096 Nov 22 06:51 etc
drwxr-xr-x  2 root root 4096 Apr 19  2012 home
drwxr-xr-x 11 root root 4096 Nov 22 06:51 lib
drwxr-xr-x  2 root root 4096 Nov 22 06:51 lib64
drwxr-xr-x  2 root root 4096 Apr 11  2013 media
drwxr-xr-x  2 root root 4096 Apr 19  2012 mnt
drwxr-xr-x  2 root root 4096 Apr 11  2013 opt
drwxr-xr-x  2 root root 4096 Apr 19  2012 proc
drwx------  2 root root 4096 Nov 22 06:51 root
drwxr-xr-x  4 root root 4096 Nov 22 06:51 run
drwxr-xr-x  2 root root 4096 Nov 22 06:51 sbin
drwxr-xr-x  2 root root 4096 Mar  5  2012 selinux
drwxr-xr-x  2 root root 4096 Apr 11  2013 srv
drwxr-xr-x  2 root root 4096 Apr 14  2012 sys
```

```
drwxrwxrwt  2 root root 4096 Apr 11  2013 tmp
drwxr-xr-x 10 root root 4096 Nov 22 06:51 usr
drwxr-xr-x 11 root root 4096 Nov 22 06:51 var
```

Pretty easy. This is the magic behind being able to refer to an image by its repository name, even if you're not interacting with the remote Docker Index or Docker Registry. Once you've pulled it down to your workstation, Docker can work with it by name using these files. This is also where things go when you're developing a new Dockerfile.

DIY Dockerfiles

Let's try an example. Make a `Dockerfile` with the following contents:

```
FROM ubuntu
```

This basically doesn't do anything except say that we're including the `ubuntu` image as our base layer, but that's enough to get started.

Next, run `docker build -t scooby_snacks .`. What that will do is look in the directory we specified (`.`) for a file called `Dockerfile` and then build it, and use the name `scooby_snacks` for the repository.

```
$ docker build -t scooby_snacks .
Uploading context 64184320 bytes
Step 1 : FROM ubuntu
 ---> 8dbd9e392a96
Successfully built 8dbd9e392a96
```

Oh no! It said *"Uploading context"*… Did we just upload it to the public registry?

Let's check:

```
$ docker search scooby_snacks
NAME      DESCRIPTION   STARS     OFFICIAL   TRUSTED
```

Whew! Not there. So why did it say that?

I have no idea, but you can ignore it. Where did it really end up?

```
$ sudo cat /var/lib/docker/repositories | python -mjson.tool
{
    "Repositories": {
        "scooby_snacks": {
            "latest": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c"
        },
        "ubuntu": {
            "12.04": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
            "12.10": "b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc",
            "latest": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
            "precise": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
            "quantal": "b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc"
        }
    }
}
```

Well, looks like Docker just *"uploaded"* it to `/var/lib/docker`. :)

It should also show up in `docker images`.

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              12.04               8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              latest              8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              precise             8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
scooby_snacks       latest              8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              12.10               b750fe79269d        8 months ago        24.65 kB (virtual 179.7 MB)
ubuntu              quantal             b750fe79269d        8 months ago        24.65 kB (virtual 179.7 MB)
```

There it is! Docker was smart enough to realize that we didn't change anything, so it kept the same image id, and didn't bother copying the `ubuntu` image. Pretty sweet.

Next, we'll make a small change so that Docker will have to build a new layer.

Edit `Dockerfile` to have these contents:

```
FROM ubuntu

RUN touch scooby_snacks.txt
```

Then run docker build -t scooby_snacks . to rebuild.

```
$ docker build -t scooby_snacks .
Uploading context 64184320 bytes
Step 1 : FROM ubuntu
 ---> 8dbd9e392a96
Step 2 : RUN touch scooby_snacks.txt
 ---> Running in 86664242766c
 ---> 91acef3a5936
Successfully built 91acef3a5936
```

There should be a new directory under /var/lib/docker/graph

```
$ sudo ls -al /var/lib/docker/graph
total 28
drwx------ 7 root root 4096 Dec 13 06:27 .
drwx------ 5 root root 4096 Dec 13 06:27 ..
drwxr-xr-x 3 root root 4096 Dec 13 04:26 27cf784147099545
drwxr-xr-x 3 root root 4096 Nov 22 06:52 8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c
drwxr-xr-x 3 root root 4096 Dec 13 06:27 91acef3a5936769f763729529e736681e5079dc6ddf6ab0e61c327a93d163df9
drwxr-xr-x 3 root root 4096 Nov 22 06:52 b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc
drwx------ 3 root root 4096 Dec 13 06:27 _tmp
```

Docker gave it a new image ID:

- 91acef3a5936769f763729529e736681e5079dc6ddf6ab0e61c327a93d163df9

It has also been updated in /var/lib/docker/repositories and docker images.

```
$ sudo cat /var/lib/docker/repositories | python -mjson.tool
{
   "Repositories": {
      "scooby_snacks": {
         "latest": "91acef3a5936769f763729529e736681e5079dc6ddf6ab0e61c327a93d163df9"
      },
      "ubuntu": {
         "12.04": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
         "12.10": "b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc",
         "latest": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
         "precise": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
         "quantal": "b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc"
      }
   }
}
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
scooby_snacks       latest              91acef3a5936         5 minutes ago        12.29 kB (virtual 131.3 MB)
```

```
ubuntu              12.04               8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              latest              8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              precise             8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              12.10               b750fe79269d        8 months ago        24.65 kB (virtual 179.7 MB)
ubuntu              quantal             b750fe79269d        8 months ago        24.65 kB (virtual 179.7 MB)
```

Let's see what`/var/lib/docker/graph/91acef3a5936769f763729529e736681e5079dc6ddf6ab0e61c327a93d163df9` looks like now:

```
$ sudo cat /var/lib/docker/graph/91acef3a5936769f763729529e736681e5079dc6ddf6ab0e61c327a93d163df9/json | python
-mjson.tool
{
    "Size": 0,
    "architecture": "x86_64",
    "config": {
        "AttachStderr": false,
        "AttachStdin": false,
        "AttachStdout": false,
        "Cmd": null,
        "CpuShares": 0,
        "Dns": null,
        "Domainname": "",
        "Entrypoint": [],
        "Env": [
            "HOME=/",
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
        ],
        "ExposedPorts": {},
        "Hostname": "86664242766c",
        "Image": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
        "Memory": 0,
        "MemorySwap": 0,
        "NetworkDisabled": false,
        "OpenStdin": false,
        "PortSpecs": null,
        "StdinOnce": false,
        "Tty": false,
        "User": "",
        "Volumes": {},
        "VolumesFrom": "",
        "WorkingDir": ""
    },
    "container": "86664242766c5548f8118716e873835c171811176a710e425c1fcf1fa367b505",
    "container_config": {
        "AttachStderr": false,
        "AttachStdin": false,
        "AttachStdout": false,
        "Cmd": [
            "/bin/sh",
            "-c",
            "touch scooby_snacks.txt"
        ],
        "CpuShares": 0,
        "Dns": null,
        "Domainname": "",
        "Entrypoint": [],
        "Env": [
            "HOME=/",
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
        ],
        "ExposedPorts": {},
        "Hostname": "86664242766c",
        "Image": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
        "Memory": 0,
        "MemorySwap": 0,
        "NetworkDisabled": false,
        "OpenStdin": false,
        "PortSpecs": null,
        "StdinOnce": false,
        "Tty": false,
        "User": "",
        "Volumes": {},
        "VolumesFrom": "",
        "WorkingDir": ""
    },
    "created": "2013-12-13T06:27:03.234029255Z",
    "docker_version": "0.6.7",
    "id": "91acef3a5936769f763729529e736681e5079dc6ddf6ab0e61c327a93d163df9",
    "parent": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c"
}
```

```
$ sudo cat /var/lib/docker/graph/91acef3a5936769f763729529e736681e5079dc6ddf6ab0e61c327a93d163df9/layersize
12288
```

```
$ sudo ls -al /var/lib/docker/graph/91acef3a5936769f763729529e736681e5079dc6ddf6ab0e61c327a93d163df9/layer
total 16
drwxr-xr-x 4 root root 4096 Dec 13 06:27 .
drwxr-xr-x 3 root root 4096 Dec 13 06:27 ..
-rw-r--r-- 1 root root    0 Dec 13 06:27 scooby_snacks.txt
-r--r--r-- 1 root root    0 Dec 13 06:27 .wh..wh.aufs
drwx------ 2 root root 4096 Dec 13 06:27 .wh..wh.orph
drwx------ 2 root root 4096 Dec 13 06:27 .wh..wh.plnk
```

Our tiny change had a big impact! Notice that Docker only kept the *differences* from the base image. This is the key to the *layer* concept.

Run it!

We can now run our new image and try it out. We'll just run an interactive `bash` prompt for now.

```
$ docker run -i -t scooby_snacks /bin/bash
root@1f8602a7d589:/# ls -al
```

```
total 12308
drwxr-xr-x  30 root root     4096 Dec 13 06:43 .
drwxr-xr-x  30 root root     4096 Dec 13 06:43 ..
-rw-------   1 root root      208 Dec 13 06:43 .dockerenv
-rwxr-xr-x   1 root root 12516574 Nov 22 02:34 .dockerinit
drwxr-xr-x   2 root root     4096 Apr 11  2013 bin
drwxr-xr-x   2 root root     4096 Apr 19  2012 boot
drwxr-xr-x   6 root root     4096 Nov 22 06:52 dev
drwxr-xr-x  41 root root     4096 Nov 22 06:52 etc
drwxr-xr-x   2 root root     4096 Apr 19  2012 home
drwxr-xr-x  11 root root     4096 Nov 22 06:51 lib
drwxr-xr-x   2 root root     4096 Nov 22 06:51 lib64
drwxr-xr-x   2 root root     4096 Apr 11  2013 media
drwxr-xr-x   2 root root     4096 Apr 19  2012 mnt
drwxr-xr-x   2 root root     4096 Apr 11  2013 opt
dr-xr-xr-x 102 root root        0 Dec 13 06:43 proc
drwx------   2 root root     4096 Nov 22 06:51 root
drwxr-xr-x   4 root root     4096 Nov 22 06:51 run
drwxr-xr-x   2 root root     4096 Nov 22 06:51 sbin
-rw-r--r--   1 root root        0 Dec 13 06:27 scooby_snacks.txt
drwxr-xr-x   2 root root     4096 Mar  5  2012 selinux
drwxr-xr-x   2 root root     4096 Apr 11  2013 srv
dr-xr-xr-x  13 root root        0 Dec 13 06:43 sys
drwxrwxrwt   2 root root     4096 Apr 11  2013 tmp
drwxr-xr-x  10 root root     4096 Nov 22 06:51 usr
drwxr-xr-x  11 root root     4096 Nov 22 06:51 var
root@1f8602a7d589:/#
```

The effects of our `RUN touch scooby_snacks.txt` command in the `Dockerfile` are exactly as expected.

Publish it!

Until now, we've been doing everything locally and not interacting with the outside world at all. This is great, we can work up a perfect `Dockerfile` before we go live. That said, I'm pretty happy with this one now, and I'm ready to publish it.

If you haven't already, make sure you make an account, and then login with `docker login`.

```
$ docker login
Username: thoward
Password:
Email: thoward37@gmail.com
Login Succeeded
```

Publish the image with `docker push scooby_snacks`

```
$ docker push scooby_snacks
2013/12/13 06:49:36 Impossible to push a "root" repository. Please rename your repository in <user>/<repo> (ex:
thoward/scooby_snacks)
```

Oops. Docker Index won't let us publish without our username in the repository name. No big deal.

Rebuild this with the correct username using

`docker build -t thoward/scooby_snacks .`

```
$ docker build -t thoward/scooby_snacks .
Uploading context 64184320 bytes
Step 1 : FROM ubuntu
 ---> 8dbd9e392a96
Step 2 : RUN touch scooby_snacks.txt
 ---> Using cache
 ---> 91acef3a5936
Successfully built 91acef3a5936
```

Nice! The message *"Using cache"* means Docker was smart enough to know that we didn't really change the image, so it didn't bother rebuilding it.

Let's try publishing again, but this time with the correct repository name:

```
$ docker push thoward/scooby_snacks
The push refers to a repository [thoward/scooby_snacks] (len: 1)
Sending image list
Pushing repository thoward/scooby_snacks (1 tags)
Pushing 8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c
Image 8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c already pushed, skipping
Pushing tags for rev [8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c] on
{https://registry-1.docker.io/v1/repositories/thoward/scooby_snacks/tags/latest}
Pushing 91acef3a5936769f763729529e736681e5079dc6ddf6ab0e61c327a93d163df9


Pushing tags for rev [91acef3a5936769f763729529e736681e5079dc6ddf6ab0e61c327a93d163df9] on
{https://registry-1.docker.io/v1/repositories/thoward/scooby_snacks/tags/latest}
```

Now that it's published, it should show up with `docker search`.

```
$ docker search scooby_snacks
NAME                 DESCRIPTION   STARS    OFFICIAL   TRUSTED
thoward/scooby_snacks              0
```

There it is. Next, let's cleanup a bit and delete the old root level one:

```
$ docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
scooby_snacks           latest             91acef3a5936        30 minutes ago      12.29 kB (virtual 131.3 MB)
thoward/scooby_snacks   latest             91acef3a5936        30 minutes ago      12.29 kB (virtual 131.3 MB)
ubuntu              12.04               8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              latest              8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              precise             8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              12.10               b750fe79269d        8 months ago        24.65 kB (virtual 179.7 MB)
ubuntu              quantal             b750fe79269d        8 months ago        24.65 kB (virtual 179.7 MB)

$ docker rmi scooby_snacks
Untagged: 91acef3a5936769f763729529e736681e5079dc6ddf6ab0e61c327a93d163df9

$ docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
thoward/scooby_snacks   latest             91acef3a5936        29 minutes ago      12.29 kB (virtual 131.3 MB)
ubuntu              12.04               8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              latest              8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              precise             8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              12.10               b750fe79269d        8 months ago        24.65 kB (virtual 179.7 MB)
ubuntu              quantal             b750fe79269d        8 months ago        24.65 kB (virtual 179.7 MB)
```

Ok, that one is gone.

Also, to be honest, this is not a very interesting image to share publicly, and we don't want to look like n00bs, so let's delete it as well.

```
$ docker rmi thoward/scooby_snacks
Untagged: 91acef3a5936769f763729529e736681e5079dc6ddf6ab0e61c327a93d163df9
Deleted: 91acef3a5936769f763729529e736681e5079dc6ddf6ab0e61c327a93d163df9

$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              12.04               8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              latest              8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              precise             8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu              12.10               b750fe79269d        8 months ago        24.65 kB (virtual 179.7 MB)
ubuntu              quantal             b750fe79269d        8 months ago        24.65 kB (virtual 179.7 MB)
```

This time, since Docker realized it was the last reference to that imageID, `docker rmi` has an additional message indicating that it deleted it instead of just 'untagging' it.

But wait! It is still public at the Docker Index, isn't it? Let's check:

```
$ docker search scooby_snacks
NAME                   DESCRIPTION   STARS     OFFICIAL   TRUSTED
thoward/scooby_snacks                0
```

Hmm.. Well this is handy, before we delete it, we can try `docker pull` and fetch it down like a "real" image and run it.

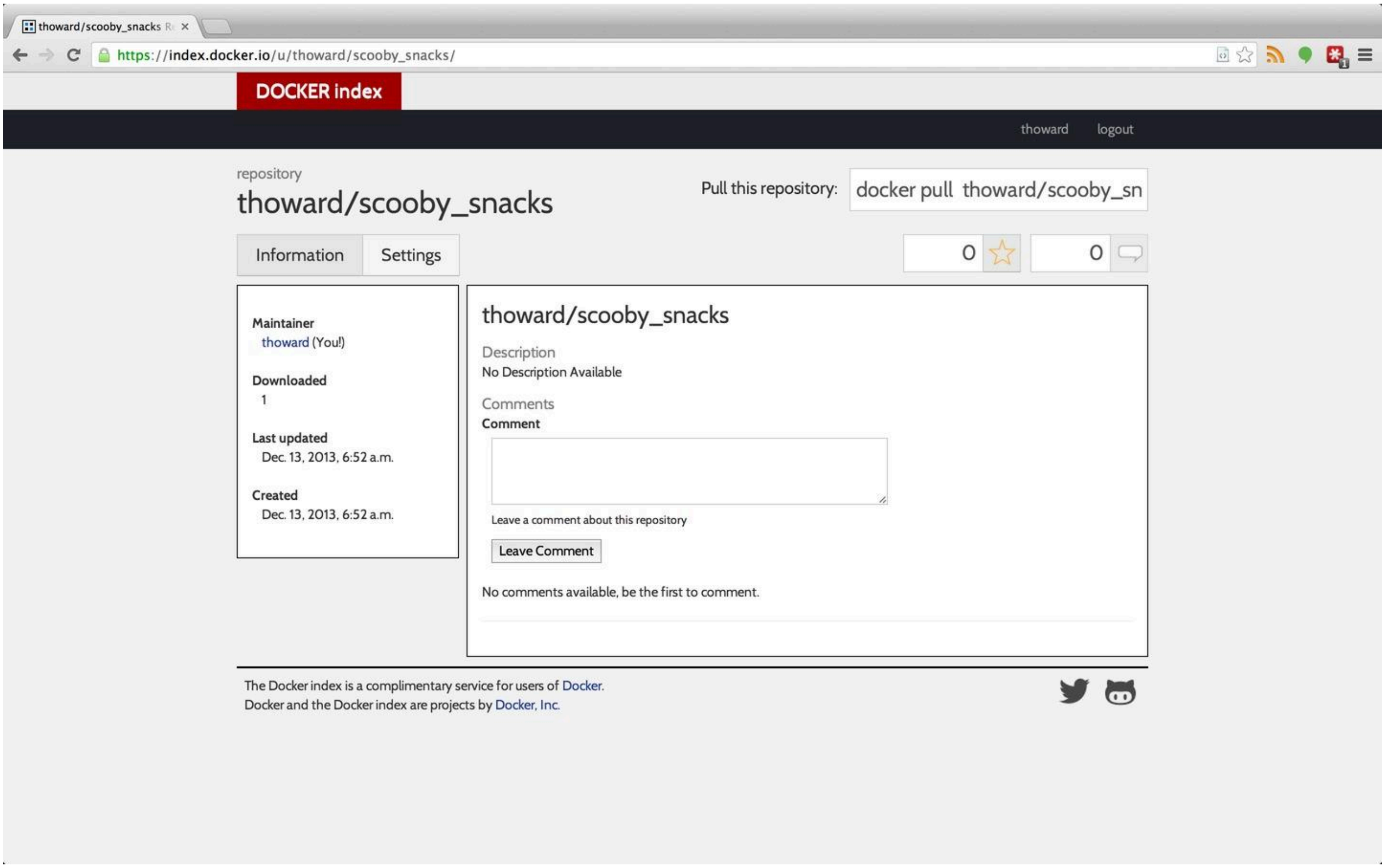
```
$ docker pull thoward/scooby_snacks
Pulling repository thoward/scooby_snacks
91acef3a5936: Download complete
8dbd9e392a96: Download complete

$ docker run -i -t thoward/scooby_snacks /bin/bash
root@90f6546bf3b7:/# ls -al
total 12308
drwxr-xr-x  30 root root     4096 Dec 13 07:03 .
drwxr-xr-x  30 root root     4096 Dec 13 07:03 ..
-rw-------   1 root root      208 Dec 13 07:03 .dockerenv
-rwxr-xr-x   1 root root 12516574 Nov 22 02:34 .dockerinit
drwxr-xr-x   2 root root     4096 Apr 11  2013 bin
drwxr-xr-x   2 root root     4096 Apr 19  2012 boot
drwxr-xr-x   6 root root     4096 Nov 22 06:52 dev
drwxr-xr-x  41 root root     4096 Nov 22 06:52 etc
drwxr-xr-x   2 root root     4096 Apr 19  2012 home
drwxr-xr-x  11 root root     4096 Nov 22 06:51 lib
drwxr-xr-x   2 root root     4096 Nov 22 06:51 lib64
drwxr-xr-x   2 root root     4096 Apr 11  2013 media
drwxr-xr-x   2 root root     4096 Apr 19  2012 mnt
drwxr-xr-x   2 root root     4096 Apr 11  2013 opt
dr-xr-xr-x 105 root root        0 Dec 13 07:03 proc
drwx------   2 root root     4096 Nov 22 06:51 root
drwxr-xr-x   4 root root     4096 Nov 22 06:51 run
drwxr-xr-x   2 root root     4096 Nov 22 06:51 sbin
-rw-r--r--   1 root root        0 Dec 13 06:27 scooby_snacks.txt
drwxr-xr-x   2 root root     4096 Mar  5  2012 selinux
drwxr-xr-x   2 root root     4096 Apr 11  2013 srv
dr-xr-xr-x  13 root root        0 Dec 13 07:03 sys
drwxrwxrwt   2 root root     4096 Apr 11  2013 tmp
drwxr-xr-x  10 root root     4096 Nov 22 06:51 usr
drwxr-xr-x  11 root root     4096 Nov 22 06:51 var
root@90f6546bf3b7:/#
```

It works!

Deleting a Published Repository

Unfortunately, to delete it from the public index/registry, we have to use the web interface, not the command-line.

First, login via the web then navigate to the repository at https://index.docker.io/u/thoward/scooby_snacks/.



repository

thoward/scooby_snacks

Pull this repository: docker pull thoward/scooby_sn

Maintainer
thoward (You!)

Downloaded
1

Last updated
Dec. 13, 2013, 6:52 a.m.

Created
Dec. 13, 2013, 6:52 a.m.

thoward/scooby_snacks

Description
No Description Available

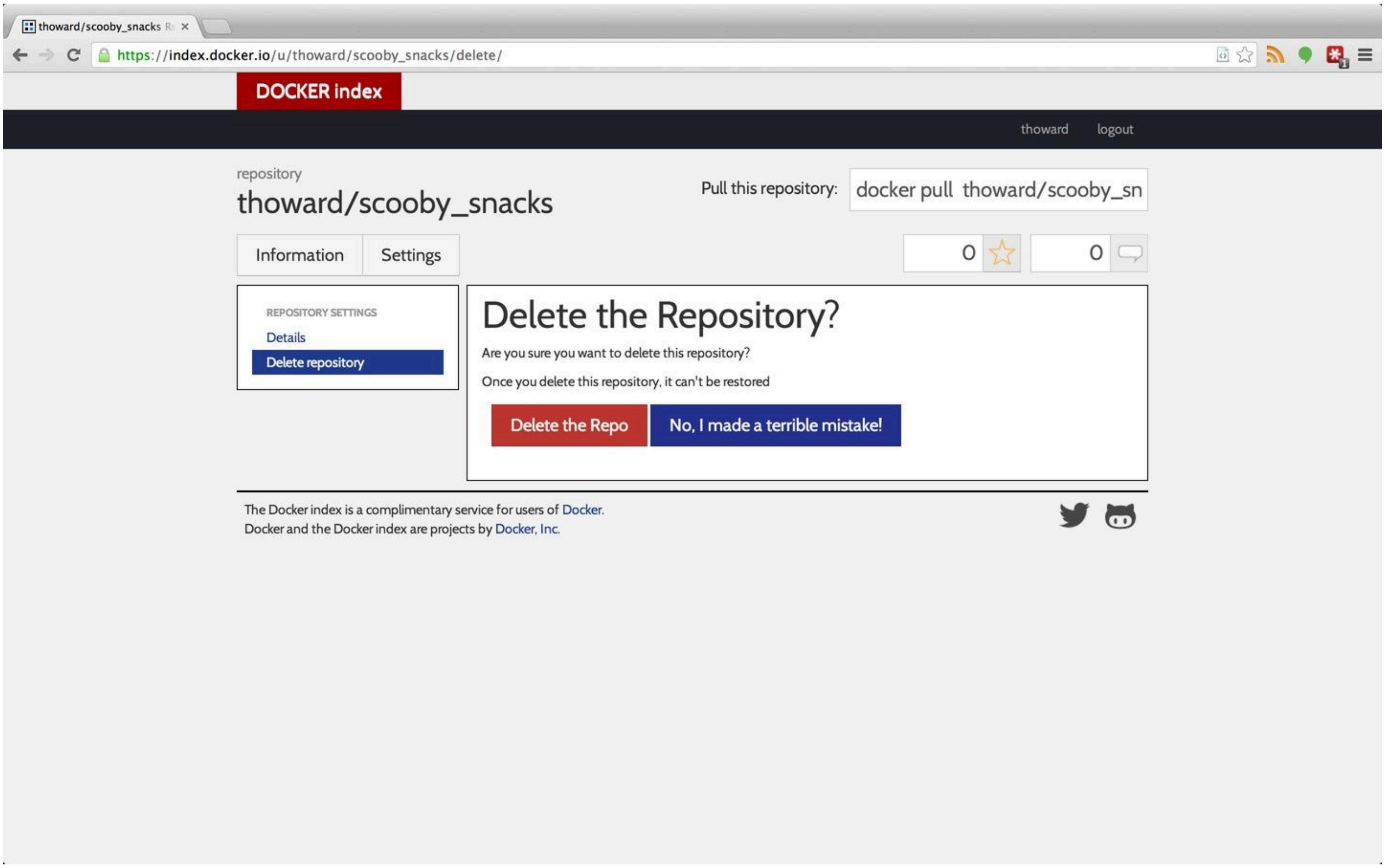
Comments
Comment

Leave a comment about this repository

No comments available, be the first to comment.

The Docker index is a complimentary service for users of Docker.
Docker and the Docker index are projects by Docker, Inc.

Click on 'Settings' tab, then 'Delete Repository' tab, then the 'Delete Repo' button.



thoward/scooby_snacks

Pull this repository: docker pull thoward/scooby_sn

Delete the Repository?

Are you sure you want to delete this repository?

Once you delete this repository, it can't be restored

The Docker index is a complimentary service for users of Docker.
Docker and the Docker index are projects by Docker, Inc.

Back on the command-line we can verify it's gone with docker search scooby_snacks

```
$ docker search scooby_snacks
NAME      DESCRIPTION   STARS     OFFICIAL   TRUSTED
```

But of course, since we never deleted the local version of it after we pulled it back down, it's still going to show up in docker images, since we have a local copy:

```
$ docker images
REPOSITORY             TAG                 IMAGE ID            CREATED             SIZE
thoward/scooby_snacks   latest              91acef3a5936        46 minutes ago      12.29 kB (virtual 131.3 MB)
ubuntu                 12.04               8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu                 latest              8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu                 precise             8dbd9e392a96        8 months ago        131.3 MB (virtual 131.3 MB)
ubuntu                 12.10               b750fe79269d        8 months ago        24.65 kB (virtual 179.7 MB)
ubuntu                 quantal             b750fe79269d        8 months ago        24.65 kB (virtual 179.7 MB)
```

So to completely remove it we need to run docker rmi again.

```
$ docker rmi thoward/scooby_snacks
Untagged: 91acef3a5936769f763729529e736681e5079dc6ddf6ab0e61c327a93d163df9
Deleted: 91acef3a5936769f763729529e736681e5079dc6ddf6ab0e61c327a93d163df9
```

Not to worry, we can always rebuild it with our `Dockerfile`. :)

Important Security Lesson

It's really important to consider the security implications of what we just saw though.

Even if a Docker image is deleted from the Docker Index *it may still be out there on someones machine*. There's no way to change that.

Also, as we saw when looking at the files we have locally, it's not quite an "opaque binary" image. All the information from the Dockerfile was in the JSON file for the image, and the artifacts of those commands are in the layer, as accessible as a filesystem. If you accidentally published a password or key, or some other critical secret, there's no getting it back, and people can find as easily as they can find anything else in a published open source code base.

Be very careful about what you're publishing. If you do accidentally publish a secret, take it down right away and update credentials on whatever systems it might have compromised.

Conclusion

Docker can be a bit confusing with its terminology, but once you wrap your head around the basic workflow described here, it should be very easy to be in-control of what you're building, knowing exactly when and how you share that with the world.