Docker 速查表

来源: docker-cheat-sheet 整理:LineZero

目录

为何使用 Docker

"通过 Docker, 开发者可以使用任何语言任何工具创建任何应用。“Dockerized” 的应用是完全可移植的，能在任何地方运行 - 不管是同事的 OS X 和 Windows 笔记本，或是在云端运行的 Ubuntu QA 服务，还是在虚拟机运行的 Red Hat 产品数据中心。

Docker Hub 上有 13,000+ 的应用，开发者可以从中选取一个进行快速扩展开发。Docker 跟踪管理变更和依赖关系，让系统管理员能更容易理解开发人员是如何让应用运转起来的。而开发者可以通过 Docker Hub 的共有/私有仓库，构建他们的自动化编译，与其他合作者共享成果。

Docker 帮助开发者更快地构建和发布高质量的应用。" -- 什么是 Docker

系统环境

我用的是 Oh My Zsh ，和 Docker 插件 ，它可以自动补全 docker 的命令。YMMV。

Linux

3.10.x 内核是能运行 Docker 的最低要求。

MacOS

10.8 “Mountain Lion” 或者更新的版本。

安装

Linux

Docker 提供了快速安装脚本：

```
curl -sSL https://get.docker.com/ | sh
```

如果你不想执行一个不明不白的 shell 脚本，那么请看安装教程，选择你在用的发行版本。

如果你是一个 Docker 超新手，那么我建议你先去看看系列教程。

Mac OS X

下载和安装 Docker Toolbox。Docker For Mac 很赞，但是它的安装和 VirtualBox 不太一样。详情请查阅比较。

> **注意** 如果你已经有安装了 docker toolbox，那么你可能会考虑通过 Docker Machine 安装包(不管是从 URL 或是 docker-machine upgrade default)升级，它确实会完成 docker-machine 的升级。但是它不会帮你升级 docker 版本 -- docker-machine 变成了 1.10.3 而 docker 还是原来的 1.8.3 或者你之前的什么版本。
>
> 所以你最好是通过 Docker Toolbox DMG 文件来升级，它会一次性的帮你处理好所有的升级。

安装好 Docker Toolbox 之后，通过 VirtualBox provider 安装带 Docker Machine 的 VM:

```
docker-machine create --driver=virtualbox default
docker-machine ls
eval "$(docker-machine env default)"
```

然后启动 container:

```
docker run hello-world
```

好了，你现在有了一个运行中的 Docker container 了。

如果你是一个 Docker 超新手，那么我建议你先去看看系列教程。

容器(Container)

最基本的 Docker 进程。容器(Container)之于虚拟机(Virtual Machine)就好比线程之于进程。或者你可以把他们想成是 chroots on steroids。

生命周期

- docker create 创建一个容器但是不启动。

- docker rename 允许重命名容器。
- docker run 在同一个操作中创建并启动一个容器。
- docker rm 删除容器。
- docker update 更新容器的资源限制。

如果你想要一个临时容器，docker run --rm 会在容器停止之后删除它。

如果你想映射宿主(host)的一个文件夹到 docker 容器，docker run -v $HOSTDIR:$DOCKERDIR。参考 Volumes。

如果你想同时删除和容器关联的 volumes ，那么在删除容器的时候必须包含 -v 选项，像这样 docker rm -v。

在 docker 1.10 中还有一个 logging driver，每个容器可以独立使用。如果你想执行 docker 并带上自定义日志驱动，这样 docker run --log-driver=syslog

启动和停止

- docker start 启动容器。
- docker stop 停止运行中的容器。
- docker restart 停止之后再启动容器。
- docker pause 暂停运行中的容器，将其 "冻结" 在当前状态。
- docker unpause 结束容器暂停状态。
- docker wait 阻塞，到运行中的容器停止为止。
- docker kill 向运行中容器发送 SIGKILL 指令。
- docker attach 链接到运行中容器。

如果你想整合容器到宿主进程管理(host process manager)，那么以 -r=false 启动守护进程(daemon)然后使用 docker start -a。

如果你想通过宿主暴露容器的端口(ports)，请看暴露端口一节。

故障 docker 实例的重启策略在这里。

CPU 限制

你可以限制 CPU，包括使用所有 CPU 的百分比，或者使用特定内核数。

比如，你可以设置 cpu-shares 。这个设置看起来有点奇怪 -- 1024 的意思是 100% CPU，因此如果你希望容器使用全体 CPU 内核的 50%，应将其设置为 512。更多信息，请查阅 https://goldmann.pl/blog/2014/09/11/resource-management-in-docker/#_cpu :

```
docker run -ti --c 512 agileek/cpuset-test
```

你可以只对某些 CPU 内核使用 cpuset-cpus]。请参阅 https://agileek.github.io/docker/2014/08/06/docker-cpuset/ 获取更多细节以及一些不错的视频:

```
docker run -ti --cpuset-cpus=0,4,6 agileek/cpuset-test
```

注意，Docker 在容器内仍然可以**看到**所有的 CPU -- 虽然它只是用了其中一部分。请查阅 https://github.com/docker/docker/issues/20770 获取更多细节。

内存限制

你同样可以在 Docker 设置内存限制：

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

能力(Capabilities)

Linux 的 capability 可以通过使用 cap-add 和 cap-drop 设置。请参阅 https://docs.docker.com/engine/reference/run/#/runtime-privilege-and-linux-capabilities 获取更多细节。这有助于提高安全性。

如需要挂载基于 FUSE 文件系统，你需要同时结合 --cap-add 和 --device 使用:

```
docker run --rm -it --cap-add SYS_ADMIN --device /dev/fuse sshfs
```

授予对单个设备访问权限:

```
docker run -it --device=/dev/ttyUSB0 debian bash
```

授予所有设备访问权限:

```
docker run -it --privileged -v /dev/bus/usb:/dev/bus/usb debian bash
```

有关容器特权的更多详情请参考这里

信息

- docker ps 查看运行中的所有容器。
- docker logs 从容器中获取日志。(你也可以使用自定义日志驱动，不过在 1.10 中，它只支持 json-file 和 journald)
- docker inspect 查看某个容器的所有信息(包括 IP 地址)。
- docker events 从容器中获取事件(events)。
- docker port 查看容器的公开端口。
- docker top 查看容器中活动进程。
- docker stats 查看容器的资源使用情况统计信息。

- docker diff 查看容器的 FS 中有变化文件信息。

docker ps -a 查看所有容器，包括正在运行的和已停止的。

docker stats --all 显示正在运行的容器列表

导入 / 导出

- docker cp 在容器和本地文件系统之间复制文件或文件夹。
- docker export 将容器的文件系统切换为压缩包(tarball archive stream)输出到 STDOUT。

执行命令

- docker exec 在容器中执行命令。

比如，进入正在运行的容器，在名为 foo 的容器中打开一个新的 shell 进程: docker exec -it foo /bin/bash.

镜像(Images)

镜像是docker 容器的模板。

生命周期

- docker images 查看所有镜像。
- docker import 从压缩文件中创建镜像。
- docker build 从 Dockerfile 创建镜像。
- docker commit 为容器创建镜像，如果容器正在运行则会临时暂停。
- docker rmi 删除镜像。
- docker load 通过 STDIN 从压缩包加载镜像，包括镜像和标签(images and tags) (0.7 起).
- docker save 通过 STDOUT 保存镜像到压缩包，包括所有的父层，标签和版本(parent layers, tags & versions) (0.7 起).

信息

- docker history 查看镜像历史记录。
- docker tag 给镜像命名打标(tags) (本地或者仓库)。

清理

虽然你可以用 docker rmi 命令来删除指定的镜像，但是这里有个称为 docker-gc 的工具，它可以以一种安全的方式，清理掉那些不再被任何容器使用的镜像。

加载/保存镜像

从文件中加载镜像:

```
docker load < my_image.tar.gz
```

保存既有镜像:

```
docker save my_image:my_tag | gzip > my_image.tar.gz
```

导入/导出容器

从文件中将容器作为镜像导入:

```
cat my_container.tar.gz | docker import - my_image:my_tag
```

导出既有容器:

```
docker export my_container | gzip > my_container.tar.gz
```

加载被保存的镜像和导入作为镜像导出的容器之间的不同

通过 load 命令来加载镜像，会创建一个新的镜像，并继承原镜像的所有历史。 通过 import 将容器作为镜像导入，也会创建一个新的镜像，但并不包含原镜像的历史，因此生成的镜像会比使用加载方式生成的镜像要小。

网络(Networks)

Docker 有网络(networks)功能。我并不是很了解它，所以这是一个扩展本文的好地方。这里有篇笔记指出，这是一种可以不使用端口来达成 docker 容器间通信的好方法。详情查阅通过网络来工作。

生命周期

- docker network create
- docker network rm

信息

- docker network ls
- docker network inspect

链接

- docker network connect
- docker network disconnect

你可以为容器指定 IP 地址:

```
# 使用你自己的子网和网关创建一个桥接网络
docker network create --subnet 203.0.113.0/24 --gateway 203.0.113.254 iptastic
```

```
# 基于以上创建的网络，运行一个nginx容器并指定ip
$ docker run --rm -it --net iptastic --ip 203.0.113.2 nginx

# 在其他地方使用curl访问这个ip（假设这是一个公网ip）
$ curl 203.0.113.2
```

仓管中心和仓库(Registry & Repository)

仓库(repository)是*被托管(hosted)*的已命名镜像(tagged images)集合，这组镜像用于构建容器文件系统。

仓管中心(registry)是一个*托管服务(host)* -- 一个服务，用于存储仓库和提供 HTTP API，以便管理上传和下载仓库。

Docker.com 把它自己的索引托管到了它的仓管中心，那里有数量众多的仓库。不过话虽如此，这个仓管中心并没有很好的验证镜像，所以如果你很担心安全问题的话，请尽量避免使用它。

- docker login 登入仓管中心。
- docker logout 登出仓管中心。
- docker search 从仓管中心检索镜像。
- docker pull 从仓管中心拉去镜像到本地。
- docker push 从本地推送镜像到仓管中心。

本地仓管中心

你可以创立一个本地的仓管中心，通过使用 docker distribution 工程，细节请查看 本地发布(local deploy) 介绍。

也可以参考 邮件列表。

Dockerfile

配置文件。当你执行 docker build 的时候会根据该配置文件设置 Docker 容器。远优于使用 docker commit。

下面是一些常用的编写 Dockerfile 的编辑器和语法高亮模块：

- 如果你使用 jEdit，我为 Dockerfile 做了个语法高亮模块。
- Sublime Text 2
- Atom
- Vim
- Emacs
- TextMate
- 如果要找更全面的关于编辑器或者 IDE 的内容，请看 当 Docker 遇上 IDE

指令

- .dockerignore
- FROM 为其他指令设置基础镜像(Base Image)。
- MAINTAINER 为生成的镜像设置作者字段。

- RUN 在当前镜像的基础上生成一个新层并执行命令。
- CMD 设置容器默认执行命令。
- EXPOSE 告知 Docker 容器在运行时所要监听的网络端口。注意：并没有实际上将端口设置为可访问。
- ENV 设置环境变量。
- ADD 将文件，文件夹或者远程文件复制到容器中。缓存无效。尽量用 COPY 代替 ADD。
- COPY 将文件或文件夹复制到容器中。
- ENTRYPOINT 将一个容器设置为可执行。
- VOLUME 为外部挂载卷标或其他容器设置挂载点(mount point)。
- USER 设置执行 RUN / CMD / ENTRYPOINT 命令的用户名。
- WORKDIR 设置工作目录。
- ARG 定义编译时(build-time)变量。
- ONBUILD 添加触发指令，当该镜像被作为其他镜像的基础镜像时该指令会被触发。
- STOPSIGNAL 设置通过系统向容器发出退出指令。
- LABEL 将键值对元数据(key/value metadata)应用到你的镜像，容器，或者守护进程。

教程

- Flux7's Dockerfile Tutorial

例子

- Examples
- Best practices for writing Dockerfiles
- Michael Crosby 还有更多的 Dockerfiles best practices / take 2
- Building Good Docker Images / Building Better Docker Images
- Managing Container Configuration with Metadata

层(Layers)

Docker 的版本化文件系统是基于层的。就像git的提交或文件变更系统一样。

注意: 如果你使用 aufs 作为你的文件系统，当删除一个容器的时候，Docker 并不一定能成功删除的文件卷标！更多详细信息请参阅 PR 8484。

链接(Links)

链接(Links)通过 TCP/IP 端口实现了 Docker 容器之间的通讯。链接到 Redis 和 Atlassian 是两个可用的例子。你还可以通过 hostname 关联链接。

注意: 如果你希望容器之间只通过链接进行通讯，在启动 docker 守护进程的时候请添加参数 -icc=false 来禁用内部进程通讯。

如果你有一个名为 CONTAINER 的容器(通过 docker run --name CONTAINER 指定) 并且在 Dockerfile 中，它的端口暴露为:

```
EXPOSE 1337
```

然后，我们创建另外一个名为 LINKED 的容器:

```
docker run -d --link CONTAINER:ALIAS --name LINKED user/wordpress
```

然后 CONTAINER 的端口和别名将会以如下的环境变量出现在 LINKED 中:

```
$ALIAS_PORT_1337_TCP_PORT
$ALIAS_PORT_1337_TCP_ADDR
```

之后你就可以通过这种方式来链接它了。

要删除链接，通过命令 docker rm --link。

通常，docker 服务之间的链接，是"服务发现"的一个子集，如果你打算在生产中大规模使用 Docker，这将是一个很大的问题。请参阅The Docker Ecosystem: Service Discovery and Distributed Configuration Stores获得更多细节。

卷标(Volumes)

Docker 的卷标(volumes)是一个free-floating 文件系统。它们不应该链接到特定的容器上。好的做法是如果可能，应当把卷标挂载到纯数据容器(data-only containers)上。

生命周期

- docker volume create
- docker volume rm

信息

- docker volume ls
- docker volume inspect

卷标在不能使用链接(只有 TCP/IP)的情况下非常有用。例如，如果你有两个 docker 实例需要通讯并在文件系统上留下记录。

你可以一次性将其挂载到多个 docker 容器上，通过 docker run --volumes-from。

因为卷标是独立的文件系统，它们通常被用于存储各容器之间的瞬时状态。也就是说，你可以配置一个无状态临时容器，关掉之后，当你有第二个这种临时容器实例的时候，你可以从上一次保存的状态继续执行。

查看卷标进阶来获取更多细节。Container42 非常有用。

你可以将宿主 MacOS 的文件夹映射为 docker 卷标:

```
docker run -v /Users/wsargent/myapp/src:/src
```

你也可以用远程 NFS 卷标，如果你觉得你有足够勇气。

可还可以考虑运行一个纯数据容器，像这里所说的那样，提供可移植数据。

暴露端口(Exposing ports)

通过宿主容器暴露输入端口是相当繁琐，但有效的。

这种方式可以将容器端口映射到宿主端口上(只使用本地主机(localhost)接口)，通过使用 -p:

```
docker run -p 127.0.0.1:$HOSTPORT:$CONTAINERPORT --name CONTAINER -t someimage
```

你可以告诉 Docker 容器在运行时监听指定的网络端口，通过使用 EXPOSE:

```
EXPOSE <CONTAINERPORT>
```

但是注意 EXPOSE 并不会暴露端口，你需要用参数 -p 。比如说你要在 localhost 上暴露容器的端口:

```
iptables -t nat -A DOCKER -p tcp --dport <LOCALHOSTPORT> -j DNAT --to-
destination <CONTAINERIP>:<PORT>
```

如果你是在 Virtualbox 中运行 Docker，那么你需要转发端口(forward the port)，使用 forwarded_port。它可以用于在 Vagrantfile 上配置暴露端口段，这样你就可以动态的映射它们了:

```
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  ...

  (49000..49900).each do |port|
    config.vm.network :forwarded_port, :host => port, :guest => port
  end

  ...
end
```

如果你忘记你将什么端口映射到宿主容器上的话，使用 docker port 来查看它:

```
docker port CONTAINER $CONTAINERPORT
```

最佳实践

这里有一些最佳实践的总结，以及一些讨论:

- The Rabbit Hole of Using Docker in Automated Tests
- Bridget Kromhout has a useful blog post on running Docker in production at Dramafever.
- There's also a best practices blog post from Lyst.
- A Docker Dev Environment in 24 Hours!
- Building a Development Environment With Docker
- Discourse in a Docker Container

安全(Security)

这节准备讨论一些关于 Docker 安全性的问题。安全这章讲述了更多细节。

首先第一件事: Docker 是有 root 权限的。如果你在 docker 组，那么你就有 root 权限。如果你暴露了 docker unix socket 给容器，意味着你赋予了容器宿主的 root 权限。Docker 不应该是你唯一的防御措施。

安全提示

为了最大的安全性，你应该会考虑在虚拟机上运行 Docker 。这是直接从 Docker 安全团队拿来的资料 -- slides / notes。然后，可以使用 AppArmor / seccomp / SELinux / grsec 之类的来限制容器的权限。更多细节，请查阅 Docker 1.10 security features。

Docker 镜像 id 属于敏感信息 所以它不应该向外界公开。你应该把他们当成密码来对待。

参考 Docker Security Cheat Sheet中 - 作者是 Thomas Sjögren - 关于如何提高容器安全的建议。

下载docker 安全测试脚本，下载白皮书 以及订阅邮件列表 (不幸的是 Docker 并没有独立的邮件列表，只有 dev / user)。

你应该远离那些使用编译版本 grsecurity / pax 的不稳定内核，比如 Alpine Linux。如果在产品中用了 grsecurity ，那么你应该考虑使用有商业支持的稳定版本，就像你对待 RedHat 那样。它要 $200 每月，对于你的运维预算来说不值一提。

从 docker 1.11 开始，你可以轻松的限制在容器中可用的进程数，以防止 fork bombs。 这要求 linux 内核 >= 4.3 并且要在内核配置中打开 CGROUP_PIDS=y 。

```
docker run --pids-limit=64
```

同时，从 docker 1.11 开始，你也可以限制进程有再获取新权限的能力了。该功能是 linux 内核从 version 3.5 开始就拥有的。你可以从这篇博客中阅读到更多关于这方面的内容。

```
docker run --security-opt=no-new-privileges
```

参考 Docker Security Cheat Sheet (它是个 PDF 版本，搞得非常难用，所以拷贝出来了) 的 容器解决方案:

关闭内部进程通讯:

```
docker -d --icc=false --iptables
```

设置容器为只读:

```
docker run --read-only
```

通过 hashsum 来验证卷标:

```
docker pull debian@sha256:a25306f3850e1bd44541976aa7b5fd0a29be
```

设置卷标为只读:

```
docker run -v $(pwd)/secrets:/secrets:ro debian
```

在 Dockerfile 中定义并运行一个用户，避免在容器中以 root 身份操作:

```
RUN groupadd -r user && useradd -r -g user user
USER user
```

用户命名空间(User Namespaces)

还可以通过使用 user namespaces -- 这已经是 1.10 内建功能了，但默认情况下是不启用的。

要在 Ubuntu 15.10 中启用用户命名空间 ("remap the userns")，请跟着这篇博客的例子来做。

安全相关视频

- Using Docker Safely
- Securing your applications using Docker
- Container security: Do containers actually contain?

安全路线图

Docker 的路线图提到关于seccomp 的支持。 这里有个 AppArmor 策略生成器，叫做 bane，他们正在实现安全配置文件。

小贴士

来源: 15 Docker Tips in 5 minutes

最后的 Ids

```
alias dl='docker ps -l -q'
docker run ubuntu echo hello world
docker commit `dl` helloworld
```

带命令行的提交 (需要 Dockerfile)

```
docker commit -run='{"Cmd":["postgres", "-too -many -opts"]}' `dl` postgres
```

获取 IP 地址

```
docker inspect `dl` | grep IPAddress | cut -d '"' -f 4
```

或者安装 jq:

```
docker inspect `dl` | jq -r '.[0].NetworkSettings.IPAddress'
```

或者用go 模板

```
docker inspect -f '{{ .NetworkSettings.IPAddress }}' <container_name>
```

获取端口映射

```
docker inspect -f '{{range $p, $conf := .NetworkSettings.Ports}} {{$p}} ->
{{(index $conf 0).HostPort}} {{end}}' <containername>
```

通过正则获取容器

```
for i in $(docker ps -a | grep "REGEXP_PATTERN" | cut -f1 -d" "); do echo $i;
done`
```

获取环境设定

```
docker run --rm ubuntu env
```

强迫关闭正在运行的容器

```
docker kill $(docker ps -q)
```

删除旧容器

```
docker ps -a | grep 'weeks ago' | awk '{print $1}' | xargs docker rm
```

删除停止容器

```
docker rm -v `docker ps -a -q -f status=exited`
```

删除 dangling 镜像

```
docker rmi $(docker images -q -f dangling=true)
```

删除所有镜像

```
docker rmi $(docker images -q)
```

删除 dangling 卷标

Docker 1.9 开始:

```
docker volume rm $(docker volume ls -q -f dangling=true)
```

1.9.0 中，过滤器 dangling=false 居然 *没* 用 - 它会被忽略然后列出所有的卷标。

查看镜像依赖

```
docker images -viz | dot -Tpng -o docker.png
```

Docker 容器瘦身 Intercity 博客

- 在当前运行层(RUN layer)清理 APT

这应当和其他 apt 命令在同一层中完成。 否则，前面的层将会保持原有信息，而你的镜像则依旧臃肿。

```
RUN {apt commands} \
  && apt-get clean \
  && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
```

- 压缩镜像

```
ID=$(docker run -d image-name /bin/bash)
docker export $ID | docker import – flat-image-name
```

- 备份

```
ID=$(docker run -d image-name /bin/bash)
(docker export $ID | gzip -c > image.tgz)
gzip -dc image.tgz | docker import - flat-image-name
```

监视运行中容器的系统资源利用率

检查某个单独容器的 CPU, 内存, 和 网络 i/o 使用情况，你可以:

```
docker stats <container>
```

按 id 列出所有的容器:

```
docker stats $(docker ps -q)
```

按名称列出所有容器:

```
docker stats $(docker ps --format '{{.Names}}')
```

按指定镜像名称列出所有容器:

```
docker ps -a -f ancestor=ubuntu
```