**[一些 Docker 的技巧与秘诀](http://blog.csdn.net/junjun16818/article/details/30696295)**

Docker 可能一开始让你望而生畏，但确实是一个非常伟大的工具。用 Shells 工作令人烦扰，并且有陷阱。鉴于我花了很多时间才弄明白它，所以我想写这篇文章来避免你们做无用功。这篇文章列举了一些快速技巧、秘诀和单行脚本，帮助你更有效地使用 Docker 。

**移除所有的容器和镜像（大扫除）**

用一行命令大扫除：

**docker** kill $(docker ps -q) ; **docker** rm $(docker ps -a -q) ; **docker** rmi $(docker images -q -a)

注：shell 中的 $() 和 `` 类似，会先执行这里面的内容，上面的脚本会出现如下 docker kill "pids" ; docker kill 在 docker 中用于停止容器，docker rm 删除容器， docker rmi 删除镜像

当没有运行的容器或者是根本没有容器的时候，这只会提示一个警告信息。当你想尝试的时候，这就是个非常好的单行命令。如果你仅仅想删除所有的容器，你可以运行如下命令：

**docker** kill $(docker ps -q) ; **docker** rm $(docker ps -a -q)

**退出时删除容器**

如果你仅仅想在一个容器中快速的运行一个命令，然后退出，并且不用担心容器状态，把 --rm 参数加入 run 命令后面,这将结束很多你保存了的容器，并且清理它们。

示例：docker run --rm -i -t busybox

/bin/bash

**不在 Shell 上运行命令**

如果你使用需要Shell 的扩展项的 docker

run 命令处理某些事情，比如 docker run --rm busybox ls '/var/log/\*', 这个命令将失败。这个失败的原因我花了工夫才弄明白。这个陷阱在这里：你原来没有 Shell ， 而 ```\* 是 Shell 的扩展项，因此你需要一个能使用的 Shell 。正确方法为：

**docker** run *--rm busybox sh -c 'ls /var/log/\*'*

**Boot2Docker 和 LapTops 处理 DNS 问题的方法**

由于这个原因， Boot2Docker 会占用 DNS 服务器很长一段时间。当你在尝试创建镜像的时候，可能会得到离谱的错误。如果你在 Ubuntu 或者 CentOS 上看到下面的提示：

cannot lookup archive.ubuntu.com

聪明的做法是停止，然后启动 boot2docker 。

**boot2docker**-cli down && boot2docker-cli up

这样问题就解决了。

**Volumes 解决 docker logs 和 docker**

**copy 问题**

如果你想在一个容器中监控另一个容器中的日志文件和文件的使用，你可以看看 [volumes](http://docs.docker.io/use/working_with_volumes/#volume-def) ,例如，检查 tomcat 是否启动：

tomcat\_id=$(docker run -d -v /var/**log**/tomcat6 wouterd/tomcat6)

*# Give Tomcat some time to wake up...*

**sleep** 5

**while** ! docker run --rm --volumes-from ${tomcat\_id} busybox /bin/sh -c "grep -i -q 'INFO: Server startup in' /var/log/tomcat6/catalina\*.log" ; **do**

echo -n "."

**sleep** 5

done

你还可以在一个 Dockerfile中指定 volumes ,这个在我前面的博客文章中结合 Docker 连载了。

**Docker Inspect 结合 Go Templates 的好处**

命令 docker inspect 允许使用 [Go Templates](http://golang.org/pkg/text/template/) 来格式化inspect 命令的输出信息如果你擅长这个，你能获取很多 docker 容器命令行的脚本输出信息。这是一个获取正在运行的容器 IP 的示例：

container\_ip=$(docker inspect --**format** '{{.NetworkSettings.IPAddress}}' ${container\_id})

这里有一个笨的技巧，用于得到匹配所有暴露(exposed)的端口 host:port ,并且把他们输入一个 java properties 文件：

sut\_ip=${BOOT\_2\_DOCKER\_HOST\_IP}

template='{{ range $key, $value := .NetworkSettings.Ports }}{{ $key }}='"${BOOT\_2\_DOCKER\_HOST\_IP}:"'{{ (index $value 0).HostPort }} {{ end }}'

tomcat\_host\_port=$(docker inspect --**format**="${template}" ${container\_id})

**for** line in ${tomcat\_host\_port} ; **do**

echo "${line}" >> ${work\_dir}/docker\_container\_hosts.properties

done

# 15 分钟掌握 15 个 Docker 小窍门

## 1. 获取最近运行容器的id

这是我们经常会用到的一个操作，按照官方示例，你可以这样做（环境**[ubuntu](http://www.21ops.com/linux/desktop" \o "ubuntu" \t "_blank)**）：

|  |  |
| --- | --- |
| 1  2  3  4 | $ ID=$(docker run ubuntu echo hello world)  hello world  $ docker commit $ID helloworld  fd08a884dc79 |

这种方式在编写脚本的时候很有用，比如你想在脚本中批量获取id，然后进一步操作。但是这种方式要求你必须给ID赋值，如果是直接敲命令，这样做就不太方便了。 这时，你可以换一种方式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $ alias dl=’docker ps -l -q’  $ docker run ubuntu echo hello world  hello world  $ dl  1904cf045887  $ docker commit `dl` helloworld  fd08a884dc79 |

docker ps -l -q命令将返回最近运行的容器的id，通过设置别名（alias），dl命令就是获取最近容器的id。这样，就无需再输入冗长的docker ps -l -q命令了。通过两个斜引号“，可以获取dl命令的值，也就是最近运行的容器的id。

## 2.尽量在Dockerfile中指定要安装的软件，而不用Docker容器的shell直接安装软件

说实话，我有时候也喜欢在shell中安装软件，也许你也一样，喜欢在shell中把所有软件安装都搞定。但是，搞来搞去，最后还是发现，你还是需要在Doockerfile中指定安装文件。在shell中安装软件，你要这样做：

|  |  |
| --- | --- |
| 1  2 | $ docker run -i -t ubuntu bash #登陆到docker容器  root@db0c3967abf8:/# |

然后输入下面的命令来安装文件：

|  |  |
| --- | --- |
| 1 | apt-get install postgresql |

然后再调用exit：

|  |  |
| --- | --- |
| 1 | root@db0c3978abf8:/# exit |

退出docker容器，再给docker commit命令传递一个复杂的JSON字符串来提交新的镜像：

|  |  |
| --- | --- |
| 1 | $ docker commit -run=”{“Cmd”:[“postgres”,”-too -many -opts”] }” `dl` postgres |

太麻烦了，不是吗？还是在Dockerfile中指定安装文件吧，只要两个步骤：

|  |  |
| --- | --- |
| 1  2 | 1.在一个小巧的Dockerfile中，指定当前操作的镜像为FROM命令的参数  2.然后在Dockerfile中指定一些docker的命令，如CMD, ENTERPOINT, VOLUME等等来指定安装的软件 |

## 3.超-超-超级用户

你可能需要一直用超级用户来操作docker，就像早期示例里一直提示的：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | # 添加docker用户组  $ sudo groupadd docker  # 把自己加到docker用户组中  $ sudo gpasswd -a myusername docker  # 重启docker后台服务  $ sudo service docker restart  # 注销，然后再登陆  $ exit |

Wow！连续三个sudo！三次化身“超级用户”，真可谓是“超-超-超级用户”啊！别担心，设置完毕，以后你就再也不用打那么多sudo了！

## 4. 清理垃圾

如果你想删除所有停止运行的容器，用这个命令：

|  |  |
| --- | --- |
| 1 | $ docker rm $(docker ps -a -q) |

顺便说一句，docker ps命令很慢，不知道为啥这么慢，按理说Go语言是很快的啊。docker ps -a -q命令列出所有容器的id，然后根据id删除容器。docker rm命令遇到正在运行的容器就会失效，所以这个命令完美的删除了所有没在运行的容器。

## 5. docker inspect输出结果的解析利器：jq

要对docker inspect的输出结果进行过滤，一般情况下，用grep命令，你需要这样操作：

|  |  |
| --- | --- |
| 1 | $docker inspect `dl` | grep IPAddress | cut -d ‘“‘ -f 4 172.17.0.52 |

哦！看上去很复杂，用jq吧，专业解析docker inspect输出结果，具有更强的可读性，方便易用：

|  |  |
| --- | --- |
| 1 | $docker inspect `dl` | jq -r ‘.[0].NetworkSettings.IPAddress’ 172.17.0.52 |

其中第一个’.’代表所有的结果。’[0]’代表数组的第一个元素。就像JavaScript访问一个JSON对象一样，简单方便。

## 6.镜像有哪些环境变量？

有时候，你需要知道自己创建的镜像有哪些环境变量。简单！只要这样：

|  |  |
| --- | --- |
| 1 | $ docker run ubuntu env |

输出结果如下：

|  |  |
| --- | --- |
| 1  2  3  4 | HOME=/  PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin  container=lxc  HOSTNAME=5e1560b7f757 |

调用env查看环境变量，对于后面要讲到的“链接”(-link)很有用，在连接两个容器时候需要用到这些环境变量，具体请看最后一个要点“链接”。

## 7.RUN命令 vs CMD命令

Docker的新手用户比较容易混淆RUN和CMD这两个命令。   
RUN命令在构建（Build）Docker时执行，这时CMD命令不执行。CMD命令在RUN命令执行时才执行。我们来理清关系，假设Dockerfile内容如下：

|  |  |
| --- | --- |
| 1  2 | FROM thelanddownunder  MAINTAINER crocdundee |

我们要向系统中安装一些软件，那么：

|  |  |
| --- | --- |
| 1  2  3  4  5 | # docker build将会执行下面的命令：  RUN apt-get update  RUN apt-get install softwares  # dokcer run默认执行下面的命令：  CMD [“softwares”] |

Build时执行RUN，RUN时执行CMD，也就是说，CMD才是镜像最终执行的命令。

## 8.CMD命令 vs ENTRYPOINT命令

又是两条容易混淆的命令！具体细节我们就不说了，举个例子，假设一个容器的Dockerfile指定CMD命令，如下：

|  |  |
| --- | --- |
| 1  2 | FROM ubuntu  CMD [“echo”] |

另一个容器的Dockerfile指定ENTRYPOINT命令，如下：

|  |  |
| --- | --- |
| 1  2 | FROM ubuntu  ENTRYPOINT [“echo”] |

运行第一个容器：

|  |  |
| --- | --- |
| 1 | docker run image1 echo hello |

得到的结果：

|  |  |
| --- | --- |
| 1 | hello |

运行第二个容器：

|  |  |
| --- | --- |
| 1 | docker run image2 echo hello |

得到的结果：

|  |  |
| --- | --- |
| 1 | echo hello |

看到不同了吧？实际上，CMD命令是可覆盖的，docker run后面输入的命令与CMD指定的命令匹配时，会把CMD指定的命令替换成docker run中带的命令。而ENTRYPOINT指定的命令只是一个“入口”，docker run后面的内容会全部传给这个“入口”，而不是进行命令的替换，所以得到的结果就是“echo hello”。

## 9.Docker容器有自己的IP地址吗？

刚接触Docker的人或许会有这样的疑问：Docker容器有自己的IP地址吗？Docker容器是一个进程？还是一个虚拟机？嗯…也许两者兼具？哈哈，其实，Docker容器确实有自己的IP，就像一个具有IP的进程。只要分别在主机和Docker容器中执行查看ip的命令就知道了。

查看主机的ip：

|  |  |
| --- | --- |
| 1 | $ ip -4 -o addr show eth0 |

得到结果：

|  |  |
| --- | --- |
| 1 | 2: eth0 inet 162.243.139.222/24 |

查看Docker容器的ip：

|  |  |
| --- | --- |
| 1 | $ docker run ubuntu ip -r -o addr show eth0 |

得到结果：

|  |  |
| --- | --- |
| 1 | 149: eth0   inet 172.17.0.43/16 |

两者并不相同，说明Docker容器有自己的ip。

## 10.基于命令行的瘦客户端，使用UNIX Socket和Docker后台服务的REST接口进行通信

Docker默认是用UNIX socket通信的，一直到大概0.5、0.6的版本还是用端口来通信，但现在则改成UNIX socket，所以从外部无法控制Docker容器的内部细节。下面我们来搞点有趣的事情，从主机链接到docker的UNIX socket：

|  |  |
| --- | --- |
| 1  2 | # 像HTTP客户端一样连接到UNIX socket  $ nc -U / /var/run/docker.sock |

连接成功后，输入：

|  |  |
| --- | --- |
| 1 | GET /images/json HTTP/1.1 |

输入后连敲两个回车，第二个回车表示输入结束。然后，得到的结果应该是：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | HTTP/1.1 200 OK  Content-Type: application/json  Date: Tue, 05 Nov 2013 23:18:09 GMT  Transfer-Encoding: chunked  16aa  [{“Repository”:”postgres”,”Tag”:”...... |

有一天，我不小心把提交的名称打错了，名字开头打成”-xxx”（我把命令和选项的顺序搞混了），所以当我删除的时候出了问题，docker rm -xxx，会把-xxx当成参数而不是镜像的名称。所以我只得通过socket直接连到容器来调用REST Server把错误的东西删掉。

## 11.把镜像的依赖关系绘制成图

docker images命令有一个很拉风的选项：-viz，可以把镜像的依赖关系绘制成图并通过管道符号保存到图片文件：

|  |  |
| --- | --- |
| 1  2 | # 生成一个依赖关系的图表  $ docker images -viz | dot -T png -o docker.png |

这样，主机的当前路径下就生成了一张png图，然后，用python开启一个微型的HTTP服务器：

|  |  |
| --- | --- |
| 1 | python -m SimpleHTTPServer |

然后在别的机器上用浏览器打开：

|  |  |
| --- | --- |
| 1 | http://machinename:8000/docker.png |

OK，依赖关系一目了然！

（译者注：要使用dot命令，主机要安装graphviz包。另外，如果主机ip没有绑定域名，machinename换成主机的ip即可。）

## 12.Docker把东西都存到哪里去了？

Docker实际上把所有东西都放到/var/lib/docker路径下了。切换成super用户，到/var/lib/docker下看看，你能学到很多有趣的东西。执行下面的命令：

|  |  |
| --- | --- |
| 1  2  3  4 | $ sudo su  # cd /var/lib/docker  # ls -F  containers/ graph/ repositories volumes/ |

可以看到不少目录，containers目录当然就是存放容器（container）了，graph目录存放镜像，文件层（file system layer）存放在graph/imageid/layer路径下，这样你就可以看看文件层里到底有哪些东西，利用这种层级结构可以清楚的看到文件层是如何一层一层叠加起来的。

## 13.Docker源代码：Go, Go, Go, Golang!

Docker的源代码全部是用Go语言写的。Go是一门非常酷的语言。其实，不只是Docker，很多优秀的软件都是用Go写的。对我来说，Docker源文件中，有4个是我非常喜欢阅读的：

##### commands.go

docker的命令行接口，是对REST API的一个轻量级封装。Docker团队不希望在命令中出现逻辑，因此commands.go只是向REST API发送指令，确保其较小的颗粒性。

##### api.go

REST API的路由（接受commands.go中的请求，转发到server.go）

##### server.go

大部分REST API的实现

##### buildfile.go

Dockerfile的解析器

有的伙计惊叹”Wow!Docker是怎么实现的？！我无法理解！”没关系，Docker是开源软件，去看它的源代码就可以了。如果你不太清楚Dockerfile中的命令是怎么回事，直接去看buildfile.go就明白了。

## 14.运行几个Docker后台程序，再退出容器，会发生什么？

OK，倒数第二个要点。如果在Docker中运行几个后台程序，再退出Docker容器，会发生什么？答案是：不要这么做！因为这样做后台程序就全丢了。

Dockerfile中用RUN命令去开启一个后台程序，如：

|  |  |
| --- | --- |
| 1 | RUN pg\_ctl start |

这样的话，RUN命令开启的后台程序就会丢失。调用容器的bash连到容器的shell：

|  |  |
| --- | --- |
| 1 | $ docker run -i -t postgresimage bash |

然后调用 ps aux查看进程，你会发现postgres的进程并没有跑起来。 RUN命令会影响文件系统。因此，不要再Dockerfile中用启动后台程序，要把后台程序启动成前台进程。或者，像一些高手提议的那样，写一个启动脚本，在脚本中启动这些后台程序或进程。

## 15.容器之间进行友好沟通：链接

这是最拉风的功能！我把它留到最后压轴！这是0.6.5中最重要的新功能，我们前面已经提过两次了。运行一个容器，给它一个名称，在下面的例子中，我们通过-name参数给容器指定名称”loldb”:

|  |  |
| --- | --- |
| 1 | $ docker run -d -name loldb loldbimage |

再运行另一个容器，加上-link参数来连接到第一个容器（别名为loldb），并给第二个容器也指定一个别名（这里用的是cheez）：

|  |  |
| --- | --- |
| 1 | $ docker run -link /loldb:cheez otherimage env |

顺便得到cheez的环境变量：

|  |  |
| --- | --- |
| 1  2  3  4  5 | CHEEZ\_PORT=tcp://172.17.0.8:6379  CHEEZ\_PORT\_1337\_TCP=tcp://172.17.0.8.6379  CHEEZ\_PORT\_1337\_TCP\_ADDR=tcp://172.17.0.12  CHEEZ\_PORT\_1337\_TCP\_PORT=6379  CHEEZ\_PORT\_1337\_TCP\_PROTO=tcp |

这样，我们就在两个容器间建立起一个网络通道（bridge），基于此，我们可以建立一个类似rails的程序：一个容器可以访问**[数据库](http://www.21ops.com/ops/dbops" \o "数据库" \t "_blank)**容器而不对外暴露其他接口。非常酷！数据库容器只需要知道第一个容器的别名（在本例中为cheez）和要打开的端口号。所以数据库容器也可以env命令来查看这个端口是否打开。