[**理解Docker（2）：Docker 镜像**](http://www.cnblogs.com/sammyliu/p/5877964.html)

本系列文章将介绍Docker的有关知识：

（1）[Docker 安装及基本用法](http://www.cnblogs.com/sammyliu/p/5875470.html)

（2）[Docker 镜像](http://www.cnblogs.com/sammyliu/p/5877964.html%20)

（3）[Docker 容器的隔离性 - 使用 Linux namespace 隔离容器的运行环境](http://www.cnblogs.com/sammyliu/p/5878973.html%20)

（4）[Docker 容器的隔离性 - 使用 cgroups 限制容器使用的资源](http://www.cnblogs.com/sammyliu/p/5886833.html%20)

（5）[Docker 网络](http://www.cnblogs.com/sammyliu/p/5894191.html%20)

  对于每个软件，除了它自身的代码以外，它的运行还需要有一个运行环境和依赖。不管这个软件是象往常一样运行在物理机或者虚机之中，还是运行在现在的容器之中，这些都是不变的。在传统环境中，软件在运行之前也需要经过 代码开发->运行环境准备 -> 安装软件 -> 运行软件 等环节，在容器环境中，中间的两个环节被镜像制作过程替代了。也就是说，镜像的制作也包括运行环境准备和安装软件等两个主要环节，以及一些其他环节。因此，Docker 容器镜像其实并没有什么新的理论，只是这过程有了新的方式而已。

  镜像(image)是动态的容器的静态表示（specification），包括容器所要运行的应用代码以及运行时的配置。Docker 镜像包括一个或者多个只读层（ read-only layers ），因此，镜像一旦被创建就再也不能被修改了。一个运行着的Docker 容器是一个镜像的实例（ instantiation ）。从同一个镜像中运行的容器包含有相同的应用代码和运行时依赖。但是不像镜像是静态的，每个运行着的容器都有一个可写层（ writable layer ，也成为容器层 container layer），它位于底下的若干只读层之上。运行时的所有变化，包括对数据和文件的写和更新，都会保存在这个层中。因此，从同一个镜像运行的多个容器包含了不同的容器层。

 Docker 有两种方式来创建一个容器镜像：

* 创建一个容器，运行若干命令，再使用 docker commit 来生成一个新的镜像。不建议使用这种方案。
* 创建一个 Dockerfile 然后再使用 docker build 来创建一个镜像。大多人会使用 Dockerfile 来创建镜像。

**1. docker build 生成镜像**

**1.1 生成过程实例**

 在使用 Dockerfile 创建容器之前，需要先准备一个 Dockerfile 文件，然后运行 docker build 命令来创建镜像。我们通过下面的例子来看看Docker 创建容器的过程。

[复制代码](javascript:void(0);)

FROM ubuntu:14.04

MAINTAINER sammy "sammy@sammy.com"

RUN apt-get update

RUN apt-get -y install ntp

EXPOSE 5555

CMD ["/usr/sbin/ntpd"]

[复制代码](javascript:void(0);)

这是一个非常简单的Dockerfile，它的目的是基于 Ubuntu 14.04 基础镜像安装 ntp 从而生成一个新的镜像。看看其过程：

[复制代码](javascript:void(0);)

root@devstack:/home/sammy/ntponubuntu# docker build -t sammy\_ntp2 .

Sending build context to Docker daemon 2.048 kB

Step 1 : FROM ubuntu:14.04

---> 4a725d3b3b1c

Step 2 : MAINTAINER sammy "sammy@sammy.com"

---> Using cache

---> c4299e3f774c

Step 3 : RUN apt-get update

---> Using cache

---> 694a19d54103

Step 4 : RUN apt-get -y install ntp

---> Running in 9bd153c65a76

Reading package lists...

...

Fetched 561 kB in 10s (51.1 kB/s)

Selecting previously unselected package libedit2:amd64.

(Reading database ... 11558 files and directories currently installed.)

...

Processing triggers for libc-bin (2.19-0ubuntu6.9) ...

Processing triggers for ureadahead (0.100.0-16) ...

---> 9cc05cf6f48d

Removing intermediate container 9bd153c65a76

Step 5 : EXPOSE 5555

---> Running in eb4633151d98

---> f5c96137bec9

Removing intermediate container eb4633151d98

Step 6 : CMD /usr/sbin/ntpd

---> Running in e81b1eae3678

---> af678df648bc

Removing intermediate container e81b1eae3678

Successfully built af678df648bc

[复制代码](javascript:void(0);)

Dockerfile 中的每个步骤都会对应每一个 docker build 输出中的 step。

Step 1：FROM ubuntu:14.04

获取基础镜像 ubuntu：14.04. Docker 首先会在本地查找，如果找到了，则直接利用；否则从 Docker registry 中下载。在第一次使用这个基础镜像的时候，Docker 会从 Docker Hub 中下载这个镜像，并保存在本地：

[复制代码](javascript:void(0);)

Step 1 : FROM ubuntu:14.04

14.04: Pulling from library/ubuntu

862a3e9af0ae: Pull complete

6498e51874bf: Pull complete

159ebdd1959b: Pull complete

0fdbedd3771a: Pull complete

7a1f7116d1e3: Pull complete

Digest: sha256:5b5d48912298181c3c80086e7d3982029b288678fccabf2265899199c24d7f89

Status: Downloaded newer image for ubuntu:14.04

---> 4a725d3b3b1c

[复制代码](javascript:void(0);)

以后再使用的时候就直接使用这个镜像而不再需要下载了。

Step 2：MAINTAINER sammy "sammy@sammy.com"

本例中依然是从 Cache 中环境新的镜像。在第一次的时候，Docker 会创建一个临时的容器 1be8f33c1846，然后运行 MAINTAINER 命令，再使用 docker commit 生成新的镜像

Step 2 : MAINTAINER sammy "sammy@sammy.com"

---> Running in 1be8f33c1846

---> c4299e3f774c

通过这个临时容器的过程（create -> commit -> destroy），生成了新的镜像 c4299e3f774c：

2016-09-16T21:58:09.010886393+08:00 container create 1be8f33c18469f089d1eee8c444dad1ff0c7309be82767092082311379245358 (image=sha256:4a725d3b3b1cc18c8cbd05358ffbbfedfe1eb947f58061e5858f08e2899731ee, name=focused\_poitras)

2016-09-16T21:58:09.060071206+08:00 container commit 1be8f33c18469f089d1eee8c444dad1ff0c7309be82767092082311379245358 (comment=, image=sha256:4a725d3b3b1cc18c8cbd05358ffbbfedfe1eb947f58061e5858f08e2899731ee, name=focused\_poitras)

2016-09-16T21:58:09.071988068+08:00 container destroy 1be8f33c18469f089d1eee8c444dad1ff0c7309be82767092082311379245358 (image=sha256:4a725d3b3b1cc18c8cbd05358ffbbfedfe1eb947f58061e5858f08e2899731ee, name=focused\_poitras)

这个镜像是基于 ubuntu 14.04 基础镜像生成的，layers 没有变化，只是元数据 CMD 发生了改变：

"Cmd": [

"/bin/sh",

"-c",

"#(nop) ",

"MAINTAINER sammy \"sammy@sammy.com\""

]

因此可以认为只是镜像的元数据发生了改变。生成的新的镜像作为中间镜像会被保存在 cache 中。

 Step 3: RUN apt-get update

本例中Docker 仍然从缓存中获取了镜像。在第一次的时候，Docker 仍然是通过创建临时容器在执行 docker commit 的方式来创建新的镜像：

[复制代码](javascript:void(0);)

Step 3 : RUN apt-get update

---> Running in 8b3b97af3bd7

Ign http://archive.ubuntu.com trusty InRelease

Get:1 http://archive.ubuntu.com trusty-updates InRelease [65.9 kB]

...

Get:22 http://archive.ubuntu.com trusty/universe amd64 Packages [7589 kB]

Fetched 22.2 MB in 16min 21s (22.6 kB/s)

Reading package lists...

---> 694a19d54103

Removing intermediate container 8b3b97af3bd7

[复制代码](javascript:void(0);)

通过以上步骤，生成了新的中间镜像 694a19d54103，它也会被保存在缓存中。你可以使用 docker inspect 694a19d54103 命令查看该中间镜像，但是无法在docker images 列表中找到它，这是因为 docker images 默认隐藏了中间状态的镜像，因此你需要使用 docker images -a 来获取它：

root@devstack:/home/sammy# docker images -a | grep 694a19d54103

<none> <none> 694a19d54103 11 hours ago 210.1 MB

该镜像和原始镜像相比，多了一个 layer，它保存的是 apt-get update 命令所带来的变化：

[复制代码](javascript:void(0);)

"RootFS": {

"Type": "layers",

"Layers": [

"sha256:102fca64f92471ff7fca48e55807ae2471502822ba620292b0a06ebcab907cf4",

"sha256:24fe29584c046f2a88f7f566dd0bf7b08a8c0d393dfad8370633b0748bba8cbc",

"sha256:530d731d21e1b1bbe356d70d3bca4d72d76fed89e90faab271d29bd58c8ccea4",

"sha256:344f56a35ff9fc747ada7d2b88bd21c49b2ec404872662cbaf0a65201873c0c6",

"sha256:ffb6ddc7582aa7e2e73f102df3ffcd272e59b7cf3f7abefe08d11a7c85dea53a",

"sha256:a1afe95c99b39c30b5c1d3e8fda451bd3f066be304616197f1046e64cf6cda93" #这一层是新加的

]

}

[复制代码](javascript:void(0);)

Step 4: RUN apt-get -y install ntp

和上面 Step 3 过程一样，这个步骤也会通过创建临时容器，执行该命令，再使用 docker commit 命令生成一个中间镜像 9cc05cf6f48d 。和上面步骤生成的镜像相比，它又多了一层：

[复制代码](javascript:void(0);)

root@devstack:/home/sammy# docker images -a | grep 9cc05cf6f48d

<none> <none> 9cc05cf6f48d 10 hours ago 212.8 MB

root@devstack:/home/sammy# docker inspect --format={{'.RootFS.Layers'}} 9cc05cf6f48d

[sha256:102fca64f92471ff7fca48e55807ae2471502822ba620292b0a06ebcab907cf4   
sha256:24fe29584c046f2a88f7f566dd0bf7b08a8c0d393dfad8370633b0748bba8cbc   
sha256:530d731d21e1b1bbe356d70d3bca4d72d76fed89e90faab271d29bd58c8ccea4   
sha256:344f56a35ff9fc747ada7d2b88bd21c49b2ec404872662cbaf0a65201873c0c6   
sha256:ffb6ddc7582aa7e2e73f102df3ffcd272e59b7cf3f7abefe08d11a7c85dea53a   
sha256:a1afe95c99b39c30b5c1d3e8fda451bd3f066be304616197f1046e64cf6cda93   
sha256:a93086f33a2b7ee18eec2454b468141f95a403f5081284b6f177f83cdb3d54ba]

[复制代码](javascript:void(0);)

Step 5: EXPOSE 5555

 这一步和上面的 Step 2 一样，Docker 生成了一个临时容器，执行 EXPOSE 55 命令，再通过 docker commit 创建了中间镜像 f5c96137bec9。该镜像的 layers 没有变化，但是元数据发生了一些变化，包括：

[复制代码](javascript:void(0);)

"ExposedPorts": {

"5555/tcp": {}

}

"Cmd": [

"/bin/sh",

"-c",

"#(nop) ",

"EXPOSE 5555/tcp"

]

[复制代码](javascript:void(0);)

Step 6: CMD ["/usr/sbin/ntpd"]

这一步和上面的步骤相同，最终它创建了镜像 af678df648bc，该镜像只是修改了 CMD 元数据：

"Cmd": [

"/bin/sh",

"-c",

"#(nop) ",

"CMD [\"/usr/sbin/ntpd\"]"

]

该镜像也是Docker 根据本 Dockerfile 生成的最终镜像。它也出现在了 docker images 结果中：

root@devstack:/home/sammy# docker images | grep af678df648bc

sammy\_ntp2 latest af678df648bc 11 hours ago 212.8 MB

我们可以使用 docker history 命令查看该镜像中每一层的信息：

[复制代码](javascript:void(0);)

root@devstack:/home/sammy/ntponubuntu# docker history af678df648bc

IMAGE CREATED CREATED BY SIZE COMMENT

af678df648bc 16 hours ago /bin/sh -c #(nop) CMD ["/usr/sbin/ntpd"] 0 B

f5c96137bec9 16 hours ago /bin/sh -c #(nop) EXPOSE 5555/tcp 0 B

9cc05cf6f48d 16 hours ago /bin/sh -c apt-get -y install ntp 2.679 MB

694a19d54103 16 hours ago /bin/sh -c apt-get update 22.17 MB

c4299e3f774c 17 hours ago /bin/sh -c #(nop) MAINTAINER sammy "sammy@sa 0 B

4a725d3b3b1c 3 weeks ago /bin/sh -c #(nop) CMD ["/bin/bash"] 0 B

<missing> 3 weeks ago /bin/sh -c mkdir -p /run/systemd && echo 'doc 7 B

<missing> 3 weeks ago /bin/sh -c sed -i 's/^#\s\*\(deb.\*universe\)$/ 1.895 kB

<missing> 3 weeks ago /bin/sh -c rm -rf /var/lib/apt/lists/\* 0 B

<missing> 3 weeks ago /bin/sh -c set -xe && echo '#!/bin/sh' > /u 194.6 kB

<missing> 3 weeks ago /bin/sh -c #(nop) ADD file:ada91758a31d8de3c7 187.8 MB

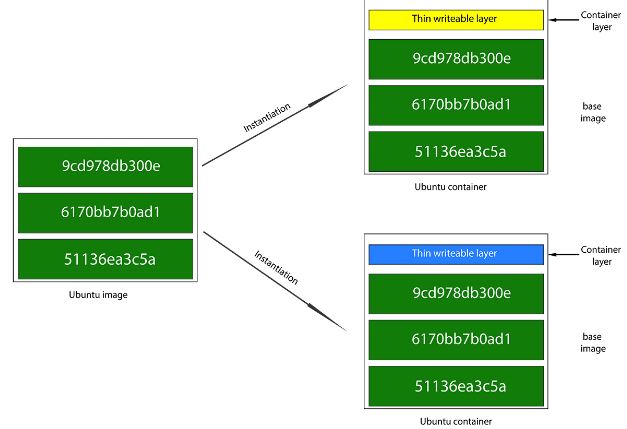
[复制代码](javascript:void(0);)

以上过程说明：

* 容器镜像包括元数据和文件系统，其中文件系统是指对基础镜像的文件系统的修改，元数据不影响文件系统，只是会影响容器的配置
* 每个步骤都会生成一个新的镜像，新的镜像与上一次的镜像相比，要么元数据有了变化，要么文件系统有了变化而多加了一层
* Docker 在需要执行指令时通过创建临时镜像，运行指定的命令，再通过 docker commit 来生成新的镜像
* Docker 会将中间镜像都保存在缓存中，这样将来如果能直接使用的话就不需要再从头创建了。关于镜像缓存，请搜索相关文档。

**1.2 Docker 镜像分层,COW 和 镜像大小（size）**

**1.2.1 镜像分层和容器层**



  从上面例子可以看出，一个 Docker 镜像是基于基础镜像的多层叠加，最终构成和容器的 rootfs （根文件系统）。当 Docker 创建一个容器时，它会在基础镜像的容器层之上添加一层新的薄薄的可写容器层。接下来，所有对容器的变化，比如写新的文件，修改已有文件和删除文件，都只会作用在这个容器层之中。因此，通过不拷贝完整的 rootfs，Docker 减少了容器所占用的空间，以及减少了容器启动所需时间。

**1.2.2 COW 和镜像大小**

  COW，copy-on-write 技术，一方面带来了容器启动的快捷，另一方也造成了容器镜像大小的增加。每一次 RUN 命令都会在镜像上增加一层，每一层都会占用磁盘空间。举个例子，在 Ubuntu 14.04 基础镜像中运行 RUN apt-get upgrade 会在保留基础层的同时再创建一个新层来放所有新的文件，而不是修改老的文件，因此，新的镜像大小会超过直接在老的文件系统上做更新时的文件大小。因此，为了减少镜像大小起见，所有文件相关的操作，比如删除，释放和移动等，都需要尽可能地放在一个 RUN 指令中进行。

  比如说，通过将上面的示例 Dockerfile 修改为：

FROM ubuntu:14.04

MAINTAINER sammy "sammy@sammy.com"

RUN apt-get update && apt-get -y install ntp

EXPOSE 5555

CMD ["/usr/sbin/ntpd"]

结果产生的镜像，不仅层数少了一层（7 -> 6），而且大小减少了 0.001M ：），因为这个例子比较特殊，文件都是添加，而没有更新，因此size 的下降非常小。

**1.2.3 使用容器需要避免的一些做法**

这篇文章 [10 things to avoid in docker containers](http://developers.redhat.com/blog/2016/02/24/10-things-to-avoid-in-docker-containers/) 列举了一些在使用容器时需要避免的做法，包括：

* 不要在容器中保存数据（**Don’t store data in containers**）
* 将应用打包到镜像再部署而不是更新到已有容器（**Don’t ship your application in two pieces**）
* 不要产生过大的镜像 （**Don’t create large images**）
* 不要使用单层镜像 （**Don’t use a single layer image**）
* 不要从运行着的容器上产生镜像 （**Don’t create images from running containers** ）
* 不要只是使用 “latest”标签 （**Don’t use only the “latest” tag**）
* 不要在容器内运行超过一个的进程 （**Don’t run more than one process in a single container** ）
* 不要在容器内保存 credentials，而是要从外面通过环境变量传入 （**Don’t store credentials in the image. Use environment variables**）
* 不要使用 root 用户跑容器进程（**Don’t run processes as a root user** ）
* 不要依赖于IP地址，而是要从外面通过环境变量传入 （**Don’t rely on IP addresses** ）

**2. Dockerfile 语法**

上面的步骤说明了 Docker 可以通过读取 Dockerfile 的内容来生成容器镜像。Dockerfile 的每一行都是 INSTRUCTION arguments 格式，即 “指令 参数”。关于 Dockerfile 的预防，请参考 <https://docs.docker.com/engine/reference/builder/>。下面只是就一些主要的指令做一些说明。

**2.1 几个主要指令**

**2.1.1 ADD 和 COPY**

Add：将 host 上的文件拷贝到或者将网络上的文件下载到容器中的指定目录

# Usage: ADD [source directory or URL] [destination directory]

ADD /my\_app\_folder /my\_app\_folder

例子：

FROM ubuntu:14.04

MAINTAINER Sammy Liu <sammy.liu@unknow.com>

ADD temp dockfile

ENTRYPOINT top

ADD 指令会将本地 temp 目录中的文件拷贝到容器的 dockfile 目录下面，从而在镜像中增加一个 layer。在未指定绝对路径的时候，会放到 WORKDIR 目录下面。

root@cc2a5605f905:/# ls dockfile/

dockerfile-add dockerfile-cmd dockerfile-env dockerfile-ports dockerfile-user dockerfile-user-h

root@cc2a5605f905:/# pwd

/

那两者有什么区别呢？

* ADD 多了2个功能, 下载URL和对支持的压缩格式的包进行解压.  其他都一样。比如 ADD http://foo.com/bar.go /tmp/main.go 会将文件从因特网上方下载下来，ADD /foo.tar.gz /tmp/ 会将压缩文件解压再COPY过去
* 如果你不希望压缩文件拷贝到container后会被解压的话, 那么使用COPY。
* 如果需要自动下载URL并拷贝到container的话, 请使用ADD

**2.1.2 CMD**

CMD：在容器被创建后执行的命令，和 RUN 不同，它是在构造容器时候所执行的命令

# Usage 1: CMD application "argument", "argument", ..

CMD "echo" "Hello docker!"

CMD 有三种格式:

* CMD ["executable","param1","param2"] (like an exec, preferred form)
* CMD ["param1","param2"] (作为 ENTRYPOINT 的参数)
* CMD command param1 param2 (作为 shell 运行)

一个Dockerfile里只能有一个CMD，如果有多个，只有最后一个生效。

**2.1.3 ENTRYPOINT**

ENTRYPOINT ：设置默认应用，会保证每次容器被创建后该应用都会被执行。CMD 和 ENTRYPOINT 的关系会在下面详细解释。

**2.1.4 ENV：设置环境变量，可以使用多次**

# Usage: ENV key value

ENV SERVER\_WORKS 4

设置了后，后续的RUN命令都可以使用，并且会作为容器的环境变量。举个例子，下面是 dockfile：

FROM ubuntu:14.04

ENV abc=1

ENV def=2

ENTRYPOINT top

生成镜像：docker build -t envimg4 -f dockerfile-env . 其元数据包括了这两个环境变量：

"Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",

"abc=1",

"def=2"

],

启动容器：docker run -it --name envc41 envimg4。也能看到：

"Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",

"abc=1",

"def=2"

]

进入容器：能看到定义的 abc 和 def 变量

root@devstack:/home/sammy/ntponubuntu# docker exec -it envc41 bash

root@ba460e0e9dc4:/# echo $abc

1

root@ba460e0e9dc4:/# echo $def

2

**2.1.5 EXPOSE ：向容器外暴露一个端口**

# Usage: EXPOSE [port]

EXPOSE 8080

**2.1.6 FROM：指定进行的基础镜像，必须是第一条指令**

# Usage: FROM [image name]

FROM ubuntu

**2.1.7 MAINTAINER：可以在任意地方使用，设置镜像的作者**

# Usage: MAINTAINER [name]

MAINTAINER authors\_name

**2.1.8 RUN：运行命令，结果会生成镜像中的一个新层**

# Usage: RUN [command]

RUN aptitude install -y ntp

**2.1.9 USER：设置该镜像的容器的主进程所使用的用户，以及后续 RUN, CMD 和 ENTRYPOINT 指令运行所使用的用户**

语法：

# Usage: USER [UID]

USER 751

Dockerfile 中的默认用户是基础镜像中所使用的用户。比如，你的镜像是从一个使用非 root 用户 sammy 的镜像继承而来的，那么你的 Dockerfile 中 RUN 指定运行的命令的用户就会使用 sammy 用户。

举例：

（1）创建 dockerfile 文件

root@devstack:/home/sammy/dockerfile# cat dockerfile-user

FROM ubuntu:14.04

USER 1000

ENTRYPOINT top

（2）创建镜像：docker build -t dockerfile-user-1000 -f dockerfile-user .

（3）启动容器：docker run -it --name c-user-1000-3 dockerfile-user-1000 top

能看出来当前用户ID 为 1000：

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

1 1000 20 0 4440 648 548 S 0.0 0.0 0:00.00 sh

5 1000 20 0 19840 1296 984 R 0.0 0.1 0:00.00 top

（4）基于该镜像再创造一个镜像，然后再启动一个容器，可以发现容器中进程所使用的用户ID 同样为 1000.

**2.1.10 VOLUME：允许容器访问host上某个目录**

# Usage: VOLUME ["/dir\_1", "/dir\_2" ..]

VOLUME ["/my\_files"]

**2.1.11 WORKDIR：设置 CMD 所指定命令的执行目录**

# Usage: WORKDIR /path

WORKDIR ~/

**2.1.12 HEALTHCHECK： 容器健康检查**

这是 Docker 1.12 版本中新引入的指令，其语法为 HEALTHCHECK [OPTIONS] CMD command。 来看一个例子：

[复制代码](javascript:void(0);)

FROM ubuntu:14.04

MAINTAINER Sammy Liu <sammy.liu@unknow.com>

RUN apt-get update

RUN apt-get -y install curl

EXPOSE 8888

CMD while true; do echo 'hello world' | nc -l -p 8888; done

HEALTHCHECK --interval=10s --timeout=2s CMD curl -f http://localhost:8888/ || exit 1

[复制代码](javascript:void(0);)

在启动容器后，其health 状态首先是 starting，然后在过了10秒做了第一次健康检查成功后，变为 healthy 状态。

root@devstack:/home/sammy/dockerfile# docker ps | grep c-health2

4c459eef1894 img-health2 "/bin/sh -c 'while tr" 7 seconds ago Up 6 seconds (health: starting) 8888/tcp c-health2

root@devstack:/home/sammy/dockerfile# docker ps | grep c-health2

4c459eef1894 img-health2 "/bin/sh -c 'while tr" 9 seconds ago Up 8 seconds (health: starting) 8888/tcp c-health2

root@devstack:/home/sammy/dockerfile# docker ps | grep c-health2

4c459eef1894 img-health2 "/bin/sh -c 'while tr" 11 seconds ago Up 11 seconds (healthy) 8888/tcp c-health2

需要注意的是 CMD 是在容器之内运行的，因此，你需要确保其命令或者脚本存在于容器之内并且可以被运行。

**2.2 几个比较绕的地方**

**2.2.1 EXPOSE 和 docker run -p -P 之间的关系**

容器的端口必须被发出（publish）出来后才能被外界使用。Dockerfile 中的 EXPOSE 只是“标记”某个端口会被暴露出来，只有在使用了 docker run -p 或者 -P 后，端口才会被“发出”出来，此时端口才能被使用。

举例：

（1）Dockerfile

FROM ubuntu:14.04

MAINTAINER Sammy Liu <sammy.liu@unknow.com>

CMD while true; do echo 'hello world' | nc -l -p 8888; done

（2）创建镜像：docker build -t no-exposed-ports -f dockerfile-ports .

（3）启动容器1：docker run -d --name no-exposed-ports1 no-exposed-ports。此容器没有 exposed 和 published 任何端口。

（4）启动容器2：docker run -d --name no-exposed-ports2 -p 8888:8888 no-exposed-ports

此时容器的 8888 端口被发布为主机上的 8888 端口：

[复制代码](javascript:void(0);)

"Ports": {

"8888/tcp": [

{

"HostIp": "0.0.0.0",

"HostPort": "8888"

}

]

}

[复制代码](javascript:void(0);)

该端口会正确返回：

root@devstack:/home/sammy/dockerfile# telnet 0.0.0.0 8888

Trying 0.0.0.0...

Connected to 0.0.0.0.

Escape character is '^]'.

hello world

Connection closed by foreign host.

（5）使用 -P 参数：docker run -d --name no-exposed-ports3 -P no-exposed-ports

此时没有任何端口被 published，说明 Docker 在使用了 “-P” 情形下只是自动将 exposed 的端口 published。

（6）使用 -p 加上一个不存在的端口：docker run -d --name no-exposed-ports4 -p 8889:8889 no-exposed-ports

此时，8889 端口会被暴露，但是没法使用。说明 -p 会将没有 exposed 的端口自动 exposed 出来。

（7）修改 dockerfile 为：

FROM ubuntu:14.04

MAINTAINER Sammy Liu <sammy.liu@unknow.com>

EXPOSE 8888

CMD while true; do echo 'hello world' | nc -l -p 8888; done

创建镜像exposed-ports， 再运行 docker run -d --name exposed-ports1 -P exposed-ports 创建一个容器，此时 8888 端口自动被 published 为主机上的 32776 端口：

[复制代码](javascript:void(0);)

"Ports": {

"8888/tcp": [

{

"HostIp": "0.0.0.0",

"HostPort": "32776"

}

]

}

[复制代码](javascript:void(0);)

可见：

* EXPOSE或者--expose只是为其他命令提供所需信息的元数据，或者只是告诉容器操作人员有哪些已知选择。它只是作为记录机制，也就是告诉用户哪些端口会提供服务。它保存在容器的元数据中。
* 使用 -p 发布特定端口。如果该端口已经被 exposed，则发布它；如果它还没有被 exposed，则它会被 exposed 和 published。Docker 不会检查容器端口的正确性。
* 使用 -P 时 Docker 会自动将所有已经被 exposed 的端口发出出来。

**2.2.2 CMD 和 ENTRYPOINT**

这两个指令都指定了运行容器时所运行的命令。以下是它们共存的一些规则：

* Dockerfile 至少需要指定一个 CMD 或者 ENTRYPOINT 指令
* CMD 可以用来指定 ENTRYPOINT 指令的参数

|  |  |  |  |
| --- | --- | --- | --- |
|  | **没有 ENTRYPOINT** | **ENTRYPOINT exec\_entry p1\_entry** | **ENTRYPOINT [“exec\_entry”, “p1\_entry”]** |
| 没有 CMD | 错误，不允许 | /bin/sh -c exec\_entry p1\_entry | exec\_entry p1\_entry |
| CMD [“exec\_cmd”, “p1\_cmd”] | exec\_cmd p1\_cmd | /bin/sh -c exec\_entry p1\_entry exec\_cmd p1\_cmd | exec\_entry p1\_entry exec\_cmd p1\_cmd |
| CMD [“p1\_cmd”, “p2\_cmd”] | p1\_cmd p2\_cmd | /bin/sh -c exec\_entry p1\_entry p1\_cmd p2\_cmd | exec\_entry p1\_entry p1\_cmd p2\_cmd |
| CMD exec\_cmd p1\_cmd | /bin/sh -c exec\_cmd p1\_cmd | /bin/sh -c exec\_entry p1\_entry /bin/sh -c exec\_cmd p1\_cmd | exec\_entry p1\_entry /bin/sh -c exec\_cmd p1\_cmd |
| 备注 | 只有 CMD 时，执行 CMD 定义的指令 | CMD 和 ENTRYPOINT 都存在时，CMD 的指令作为 ENTRYPOINT 的参数 |  |

 举例：

（1）同时有 CMD 和 ENTRYPOINT

FROM ubuntu:14.04

MAINTAINER Sammy Liu <sammy.liu@unknow.com>

CMD top

ENTRYPOINT ps

此时会运行的指令为 /bin/sh -c ps /bin/sh -c top

但是实际上只是运行了 ps：

[复制代码](javascript:void(0);)

root@devstack:/home/sammy/dockerfile# /bin/sh -c ps /bin/sh -c top

PID TTY TIME CMD

10789 pts/3 00:00:00 su

10790 pts/3 00:00:00 bash

18479 pts/3 00:00:00 sh

18480 pts/3 00:00:00 ps

root@devstack:/home/sammy/dockerfile# /bin/sh -c ps

PID TTY TIME CMD

10789 pts/3 00:00:00 su

10790 pts/3 00:00:00 bash

18481 pts/3 00:00:00 sh

18482 pts/3 00:00:00 ps

[复制代码](javascript:void(0);)

（2）CMD 作为 ENTRYPOINT 的参数

FROM ubuntu:14.04

MAINTAINER Sammy Liu <sammy.liu@unknow.com>

CMD ["-n", "10"]

ENTRYPOINT top

启动容器后运行的命令为 /bin/sh -c top -n 10.

参考链接

* <http://developers.redhat.com/blog/2016/03/09/more-about-docker-images-size/>
* <http://developers.redhat.com/blog/2016/02/24/10-things-to-avoid-in-docker-containers/>