

深入理解 Docker 镜像 json 文件

「Allen 谈 Docker 系列」DaoCloud 正在启动 Docker 技术系列文章，每周都会为大家推送一期真材实料的精选 Docker 文章。主讲人为 DaoCloud 核心开发团队成员 Allen 孙宏亮，他是 InfoQ《Docker 源码分析》专栏作者，已出版《Docker 源码分析》一书。Allen 接触 Docker 近两年，爱钻研系统实现原理，及 Linux 操作系统。

很多时候，当大家谈论起 Docker，经常会提到 Docker 作为容器解决方案，在虚拟化资源方面存在不小优势。轻量级虚拟化技术的优点暂且不谈，从软件生命周期来看，Docker 在打包软件、分发软件方面的能力同样出众。而后者很大程度上依赖于 Docker 的镜像技术。

Docker 镜像技术提供了一套标准，创造性地使用 Dockerfile 来规范化 Docker 化应用的制作流程，结果产生的 Docker 镜像便于传输与管理，最终通过 Docker 镜像运行 Docker 容器，完成容器化应用的交付。

经过本系列对于 Docker 镜像的镜像，大家应该已经清楚 Docker 镜像的存储、以及 Docker 镜像的内容。Docker 镜像的内容，应该说包含两部分，除了镜像层中的文件之外，还包括一层镜像的 json 文件。镜像层文件的理解较为简单，但是 Docker 镜像的 json 文件理解起来就会稍显复杂。本文就带大家深入理解 Docker 镜像的 json 文件。

我们一直提到“通过 Docker 镜像运行 Docker 容器”，如果仔细思考这句话，可能依然会存在一些疑惑。不难的理解是：Docker 镜像层中的文件全部属于静态的磁盘文件，而 Docker 容器属于一个动态的产物，可以认为是一个或者多个运行中的进程。那么，静态的 Docker 镜像转换为动态的 Docker 容器背后肯定会有一些不为人知的秘密。我们不妨带着以下几个问题来思考 Docker 镜像的 json 文件：

1. 如何判定一个 Docker 镜像应该运行哪个进程，这部分信息存在哪？

2. 有了以上信息，将 Docker 镜像运行成 Docker 容器的行为是谁在主导？

镜像的静态与容器的动态

这一次，我们依旧从动态和静态这两个词来看 Docker 镜像的前世今生。首先，Docker 镜像的镜像层文件属于静态文件，当容器运行起来之后这部分内容将作为 Docker 容器的文件系统内容，提供 Docker 容器的文件系统视角。我们带着这样的观点，再来看 Dockerfile 的概念。

在 Dockerfile 的原语中，大家肯定对 ENV、VOLUME、EXPOSE 以及 CMD 等命令非常熟悉。

ENV MYPATH=/root: ENV 命令在构建 Docker 镜像时，为镜像添加一个环境变量，以便该环境变量在启动 Docker 容器时作用于容器内的进程；这部分信息不应该以静态文件的形式被打入 Docker 的镜像层文件。

VOLUME /data: VOLUME 命令在构建 Docker 镜像时，为镜像添加一个数据卷标识，以便通过该镜像运行容器时为容器挂载一个数据卷；由于构建时真实的数据卷还不存在，所以这部分信息不应该以静态文件的形式被打入 Docker 的镜像层文件。

EXPOSE 80: EXPOSE命令在构建 Docker 镜像时，记录容器内部实际监听的端口，以便在通过 bridge 模式配置 Docker 容器网络时，将该端口与宿主机进行一次 DNAT 转换；这部分信息属于 Docker 容器运行时所需要的信息，也不应该以静态文件的形式被打入 Docker 的镜像层文件。

CMD ["/run.sh"]: CMD 命令在构建 Docker 镜像时，记录启动 Docker 容器的执行命令入口，一般用以指定用户的应用程序；这部分配置信息更不应该以静态文件的形式被打入 Docker 的镜像层文件。

Dockerfile 中以上举例的4类命令，通过分析，我们得出初步的结论：Dockerfile的部分命令各自包含一类动态信息，这类信息不属于 Docker 镜像层中的文件内容。

动态内容的存储

高移植性保障了 Docker 镜像的一次构建，多处运行，那么使得 Docker 容器可以顺利运行，Docker 自然不会抛弃构建 Docker 镜像时的动态内容。因此，动态内容的存储就显得尤为重要。

此时就是Docker 镜像 json 文件登场的时机。构建 Docker 镜像时，所有动态的信息都会记录进相应 Docker 镜像的 json 文件中。

需要注意的是，虽然镜像的动态信息会被存储于 Docker 镜像的 json 文件中，但是并不代表 json 文件中仅存储动态信息，Dockerfile 构建过程中，机会所有的操作都会记录在 json 文件中。

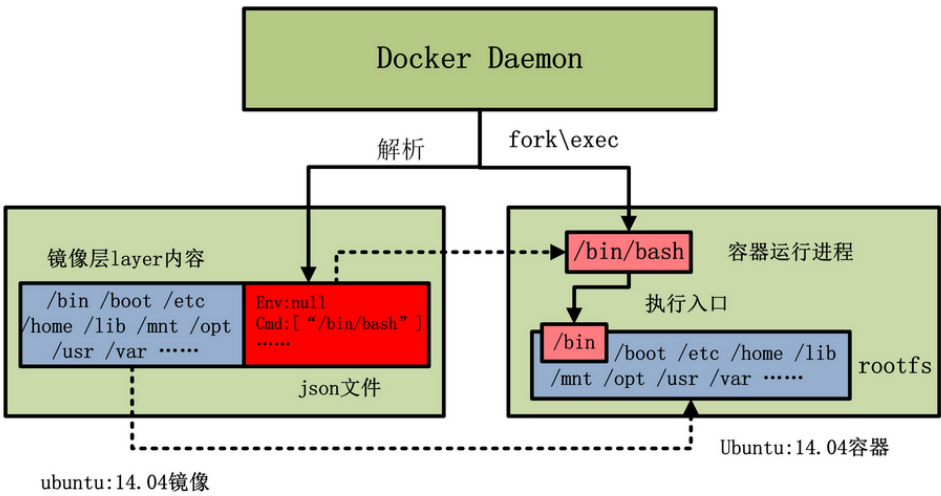
动态内容的执行

有了 Docker 镜像 json 文件来描述 Docker 容器的动态信息，那么 json 文件作为 Docker 镜像的一部分，在 Docker 体系中，由哪一模块来完成 json 文件中动态信息的解析与执行呢？

如果大家清楚“每一个 Docker 容器都是 Docker Daemon 的子进程”的话，肯定会联想到 Docker Daemon。答案也正是 Docker Daemon。站在启动容器的角度上，Docker Daemon 的作用就是以下两点：

- 1.将 Docker 镜像的镜像层文件作为 Docker 容器的 rootfs。
- 2.提取 Docker 镜像 json 文件中的动态文件，确定启动进程，并为之配置动态运行环境。

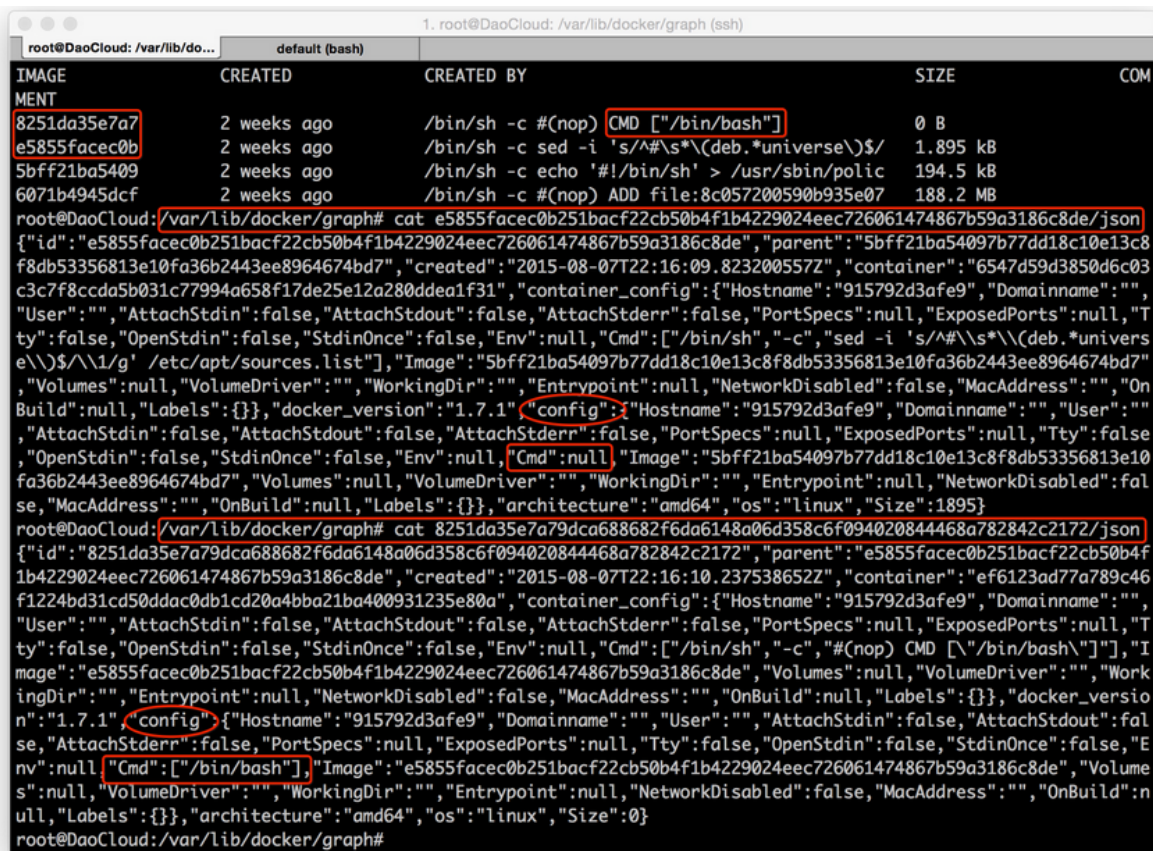
Docker Daemon、Docker 镜像以及 Docker 容器三者的简单示意图如下：



通过上图，我们可以使用 ubuntu:14.04 镜像运行 Docker 容器时，前者的镜像层 layer 内容将作为容器的 rootfs；而前者的 json 文件，会由 Docker Daemon 解析，并提取出其中的容器执行入口 CMD 信息，以及容器进程的环境变量 ENV 信息，最终初始化容器进程。当然，容器进程的执行入口来源于镜像提供的 rootfs。假如此时 ubuntu 14.04 镜像的 json 文件中又含有 VOLUME 信息，那么 Docker Daemon 将会为 Docker 容器在宿主机上创建一个文件目录，并挂载到容器内部，实现镜像中 VOLUME 动态信息的运用，其他动态信息的运用也大同小异。

Docker 镜像 json 文件的真面目

全文分析至此，还是更多的从理论的角度阐述 Docker 镜像的 json 文件，那么现实情况中，此类 json 文件到底存的内容是什么呢？我们依然以 ubuntu:14.04 为例，揭开 Docker 镜像 json 文件的这面目。



```
1. root@DaoCloud: /var/lib/docker/graph (ssh)
root@DaoCloud: /var/lib/do... default (bash)
IMAGE          CREATED        CREATED BY          SIZE            COM
MENT
8251da35e7a7   2 weeks ago   /bin/sh -c #(nop)  CMD ["/bin/bash"]   0 B
e5855facec0b   2 weeks ago   /bin/sh -c sed -i 's/^#\s*(deb.*universe\)$/ /usr/sbin/polic 1.895 kB
5bfff21ba5409 2 weeks ago   /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic 194.5 kB
6071b4945dcf   2 weeks ago   /bin/sh -c #(nop)  ADD file:8c057200590b935e07 188.2 MB

root@DaoCloud: /var/lib/docker/graph# cat e5855facec0b251bacf22cb50b4f1b4229024eec726061474867b59a3186c8de/json
{"id": "e5855facec0b251bacf22cb50b4f1b4229024eec726061474867b59a3186c8de", "parent": "5bfff21ba54097b77dd18c10e13c8f8db53356813e10fa36b2443ee8964674bd7", "created": "2015-08-07T22:16:09.823200557Z", "container": "6547d59d3850d6c03c3c7f8ccda5b031c77994a658f17de25e12a280ddeaf131", "container_config": {"Hostname": "915792d3afe9", "Domainname": "", "User": "", "AttachStdin": false, "AttachStdout": false, "AttachStderr": false, "PortSpecs": null, "ExposedPorts": null, "Tty": false, "OpenStdin": false, "StdinOnce": false, "Env": null, "Cmd": ["/bin/sh", "-c", "sed -i 's/^#\s*(deb.*universe\)$/ /usr/sbin/polic /etc/apt/sources.list"], "Image": "5bfff21ba54097b77dd18c10e13c8f8db53356813e10fa36b2443ee8964674bd7", "Volumes": null, "VolumeDriver": "", "WorkingDir": "", "Entrypoint": null, "NetworkDisabled": false, "MacAddress": "", "OnBuild": null, "Labels": {}, "docker_version": "1.7.1", "config": {"Hostname": "915792d3afe9", "Domainname": "", "User": "", "AttachStdin": false, "AttachStdout": false, "AttachStderr": false, "PortSpecs": null, "ExposedPorts": null, "Tty": false, "OpenStdin": false, "StdinOnce": false, "Env": null, "Cmd": null, "Image": "5bfff21ba54097b77dd18c10e13c8f8db53356813e10fa36b2443ee8964674bd7", "Volumes": null, "VolumeDriver": "", "WorkingDir": "", "Entrypoint": null, "NetworkDisabled": false, "MacAddress": "", "OnBuild": null, "Labels": {}, "architecture": "amd64", "os": "linux", "Size": 1895}

root@DaoCloud: /var/lib/docker/graph# cat 8251da35e7a79dca688682f6da6148a06d358c6f094020844468a782842c2172/json
{"id": "8251da35e7a79dca688682f6da6148a06d358c6f094020844468a782842c2172", "parent": "e5855facec0b251bacf22cb50b4f1b4229024eec726061474867b59a3186c8de", "created": "2015-08-07T22:16:10.237538652Z", "container": "ef6123ad77a789c46f1224bd31cd50ddac0db1cd20a4bba21ba400931235e80a", "container_config": {"Hostname": "915792d3afe9", "Domainname": "", "User": "", "AttachStdin": false, "AttachStdout": false, "AttachStderr": false, "PortSpecs": null, "ExposedPorts": null, "Tty": false, "OpenStdin": false, "StdinOnce": false, "Env": null, "Cmd": ["/bin/sh", "-c", "#(nop) CMD [\"/bin/bash\"]"], "Image": "e5855facec0b251bacf22cb50b4f1b4229024eec726061474867b59a3186c8de", "Volumes": null, "VolumeDriver": "", "WorkingDir": "", "Entrypoint": null, "NetworkDisabled": false, "MacAddress": "", "OnBuild": null, "Labels": {}, "docker_version": "1.7.1", "config": {"Hostname": "915792d3afe9", "Domainname": "", "User": "", "AttachStdin": false, "AttachStdout": false, "AttachStderr": false, "PortSpecs": null, "ExposedPorts": null, "Tty": false, "OpenStdin": false, "StdinOnce": false, "Env": null, "Cmd": ["/bin/bash"], "Image": "e5855facec0b251bacf22cb50b4f1b4229024eec726061474867b59a3186c8de", "Volumes": null, "VolumeDriver": "", "WorkingDir": "", "Entrypoint": null, "NetworkDisabled": false, "MacAddress": "", "OnBuild": null, "Labels": {}, "architecture": "amd64", "os": "linux", "Size": 0}

root@DaoCloud: /var/lib/docker/graph#
```

一个含有标签的 Docker 镜像一般会有一个或者多个层级镜像组合而成，而每个镜像层都会含有一个 json 文件。上图中，我们展现了 ubuntu:14.04 镜像中 4 个镜像层的具体情况，特别分析了镜像8251da35e7a7和e5855facec0b。通过查看这两个镜像的 json 文件，我们可以发现，两个 json 文件中的 config 属性中，除了理所应当不同的镜像 ID 之外，只有 Cmd 属性不同。由于镜像e5855facec0b是镜像8251da35e7a7的父镜像，同时构建子镜像的时候使用的 Dockerfile 命令为CMD ["/bin/bash"]，因此子镜像在父镜像 json 文件的基础上，更新 config 属性中的 Cmd 属性，完成自身 json 文件的生成。Docker 镜像中父子镜像的 json 文件有很大的相似性，子镜像仅在父镜像 json 文件的基础上，修改运行自身对应的 Dockerfile 命令后造成的差异。

镜像构建完毕，Docker 镜像的镜像层 layer 文件以及 json 文件均准备完毕，那么当 Docker Daemon 通过此 ubuntu:14.04 镜像运行 Docker 容器时，首先提取最上层镜像的 json 文件，找到 config 属性中的 Cmd 命令，并在镜像层文件构成的容器 rootfs 中找到

相应的执行程序，最终执行，完成容器的启动。（实际情况要更复杂，会涉及 Entrypoint 以及 Cmd 两部分的内容，本系列后续会深入深入两者的作用与区别）

当然，除了 Cmd 之外，json 文件的 config 属性中同时还存在 User、ExposedPorts、Entrypoint、NetworkDisabled 等一系列启动容器时所需的动态信息。

总结

Docker 镜像的 json 文件扮演极其重要的角色，提供了静态镜像向动态容器的转化依据，同时清晰地记录了父子镜像之间的差异，基于此差异，后续 Docker 构建的 cache 机制才得以实现。

欲知 Docker 镜像更精彩的内容，且听下回分解。下回内容预告：docker build 的 cache 机制分析

Allen 孙宏亮

硕士，浙江大学毕业，现为 DaoCloud 软件工程师，出版有《Docker 源码分析》，目前主要负责企业级容器云平台的研发工作。数年来一直从事云计算、PaaS 领域的研究与实践，是国内较早一批接触 Docker 的先行者，同时也是 Docker 技术的推广者。

分享给朋友

Python 开发者的 Docker 之旅

「人生苦短，我用 Python」这句话作为「Docker 开发大礼包」的第二季的开篇引言是再合适不过了。这句话的出处是...

与 Docker 和 DaoCloud 共舞

[编者的话] 当文字偶遇代码，当程序插上了翅膀，让分享成为我们彼此沟通的语言。我们期待可以构建这样一个平台让开发者们看到你们的智慧，挖掘你们的才华，让彼此在开源的路上不再孤独。 “ DaoCloud...