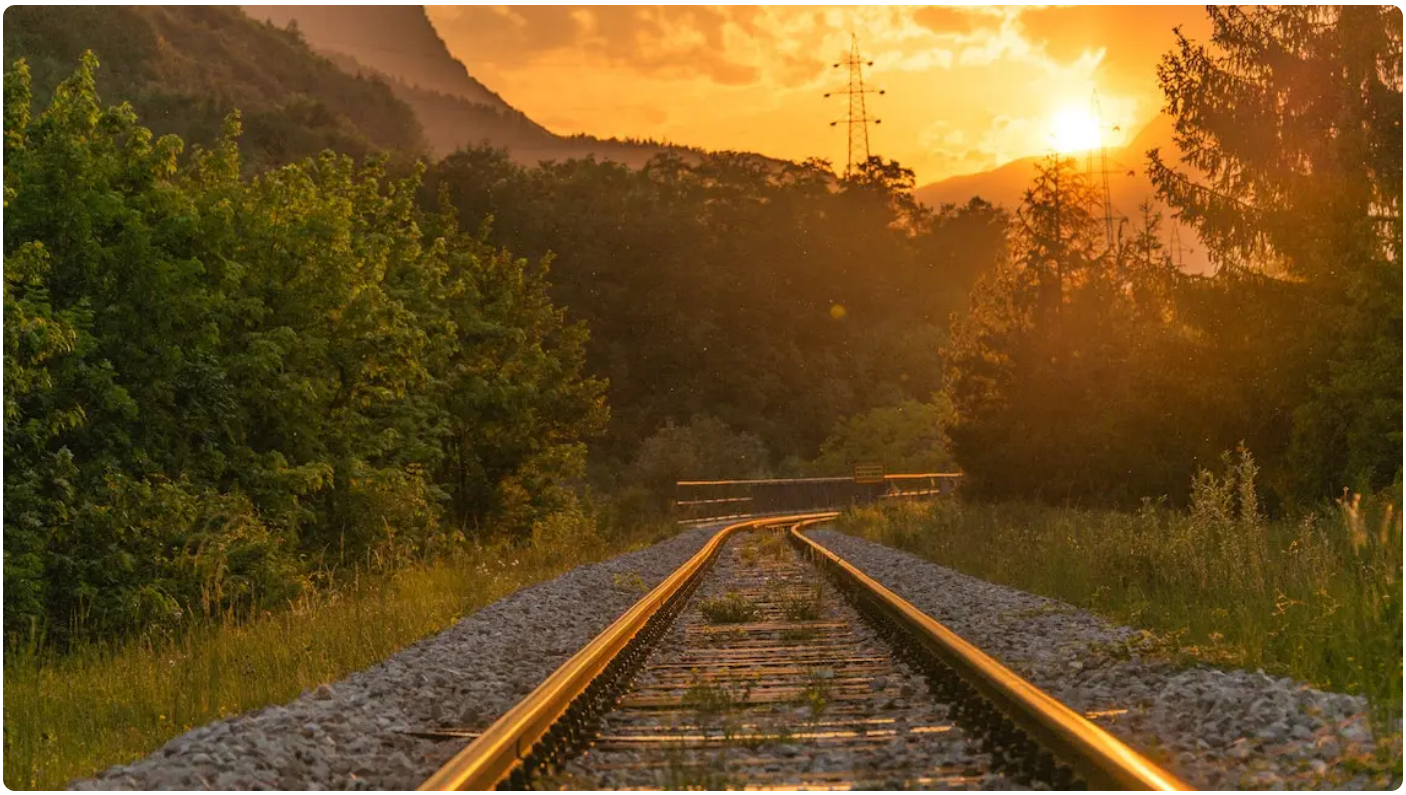
01 | 如何将业务代码构建为容器镜像？

2022-12-12 王炜 来自北京

《云原生架构与GitOps实战》

课程介绍 >



讲述：王炜

时长 14:18　大小 13.06M

你好，我是王炜。从这节课开始，我们正式进入专栏的学习。这节课我们重点来看一下，如何将业务代码构建为容器镜像。

在工作中，我相信你经常会从同事那里听到这几个熟悉的词：Docker、镜像、镜像仓库、容器等等。

紧接着，你脑海中会浮现出几个问题：容器和镜像之间的关系是什么？如何将业务代码构建为容器镜像？容器镜像又是怎么存储和使用的呢？

别着急，这节课，我会带你从 0 开始认识作为云原生基石的容器镜像，让你在实践中理解镜像和容器的概念，也在这个过程中构建你的第一个容器镜像。

初识容器镜像

在开始实践之前，你需要准备一台电脑，推荐 Linux 或者 macOS，并安装好 Docker，具体流程你可以参考官网。Windows 系统也同样适用，只是需要注意一些操作上的差异。

接下来，我们在本地拉取一个镜像并将它运行起来，看看镜像到底能做什么。

我们首先要用下面这个命令从官方镜像仓库中拉取一个镜像到本地，这是我提前制作好的演示镜像。

```
$ docker pull lyzhang1999/hello-world-flask:latest
```

提醒一下，如果拉取镜像失败，推荐你开通一台国内云厂商的香港 Linux 主机进行接下来的实验。

这里要注意两个细节。第一，这里并没有指定完整的镜像地址，Docker 会默认从 docker.io 官方镜像仓库中搜索。所以，你可以理解为，上面这段代码的 lyzhang1999/hello-world-flask:latest 和 docker.io/lyzhang1999/hello-world-flask:latest 是等效的。第二个细节是，冒号后面的 latest 指的是镜像版本号。

那么我们怎么查看本地已经拉取了哪些镜像呢？你可以使用 docker images 命令来查看它们：

```
$ docker images
REPOSITORY                          TAG        IMAGE ID       CREATED          SIZE
lyzhang1999/hello-world-flask                  latest     e2b1a18ed1c1   1 minutes
```

显然，这里打印的结果就是我们刚才拉取的镜像。

接下来，到了最重要的一步：**运行镜像。**我们可以使用 docker run 命令来运行镜像。

```
$ docker run -d -p 8000:5000 lyzhang1999/hello-world-flask:latest
c370825640b6b3669cae20f14e2684ec82b20e4980b329c02b47e47771c931fd
```

看到上面的输出说明我们成功启动了 hello-world-flask 镜像。

-d 代表“在后台运行容器”，同时它会输出容器 ID，这是运行容器的唯一标识。

-p 代表“将容器内的 5000 端口暴露到宿主机（本地的 8000 端口）”，这可以方便我们在本地进行访问。

现在，我们打开浏览器访问 localhost:8000，可以看到下面这段输出内容：

```
Hello, my first docker images!
```

通过这么简单的几条命令，我们就完成了从拉取镜像到运行镜像的全过程。怎么样，是不是比想象中更加简单？

让我们继续探究这个正在运行中的容器。**我们尝试进入容器内部。**

首先，你可以使用 docker ps 命令来查看当前运行中的容器列表，输出结果的“c370825640b6” 即为容器 ID：

```
$ docker ps
CONTAINER ID   IMAGE                                  COMMAND
c370825640b6   lyzhang1999/hello-world-flask:latest   "python3 -m flask ru…"
```

然后，我们使用 docker exec [容器 ID] 进入容器内部：

```
$ docker exec -it c370825640b6 bash
root@c370825640b6:/app#
```

-it 的含义是“保持 STDIN 打开状态，并且分配一个虚拟的终端（Terminal）”。你可以简单理解为，我们通过 SSH 登录到了容器内部，在当前终端下运行的所有命令都是基于容器内的。

例如，你可以在当前终端执行 ls 查看容器内的文件：


```
$ root@c370825640b6:/app# ls
Dockerfile  __pycache__  app.py  requirements.txt
```

从上面的返回结果来看，容器内的工作目录 /app 包含了我们之前从本地复制到镜像内的 Dockerfile、app.py 和 requirements.txt 三个文件，这和我们对 Dockerfile 的 COPY 指令预期的行为是一致的。

现在，我们尝试着编辑容器内的 app.py。

复制代码

```
$ vi app.py
```

修改第 8 行的输出并保存，重新刷新浏览器，可以看到我们对容器内的修改实时生效了。

最后，我们可以像退出 SSH 登录一样，在容器的终端下执行 exit 命令退出，返回宿主机也就是我们本机的终端。

复制代码

```
$ root@c370825640b6:/app# exit
```

列出和编辑文件的操作，事实上都是对容器的操作。我刚开始学习的时候，经常把镜像和容器这两个概念搞混。

通俗地说，镜像是一个同时包含业务应用和运行环境的“系统安装包”，它需要运行起来之后才能提供服务，运行后镜像的“实例化”称为容器（Container）。你可以对同一个镜像实例化多次，产生多个独立的容器，这些容器拥有不同的容器 ID，不同的容器之间是相互隔离的。

进一步理解，你可以把容器比喻为虚拟机，虚拟机也是，彼此之间的数据和状态都是隔离的。

最后，要想停止运行中的容器呢，只需要使用 docker stop [容器 ID] 命令就可以了。

```
$ docker stop c370825640b6
```


到这里，我相信你对镜像和容器已经有了基本的认识。

镜像是怎么构建出来的？

但镜像到底是怎么被构建出来的呢？我的业务代码如何打包成镜像？接下来，我就带你从 0 开始构建你的第一个镜像。

这里我使用 Python 编写的 Flask Web 应用作为例子。**首先，假设这是我的业务代码，请你将下面这段代码保存为 app.py。**

```python
from flask import Flask
import os
app = Flask(__name__)
app.run(debug=True)

@app.route('/')
def hello_world():
    return 'Hello, my first docker images! ' + os.getenv("HOSTNAME") + ''
```

这段代码的含义非常简单，启动一个 Web 服务器，当接收到 HTTP 请求时，返回 “Hello, my first docker images!” 以及 HOSTNAME 环境变量。

接下来，我们创建 Python 的依赖文件 requirements.txt ，用它来安装我们所依赖的 Flask 框架。你可以执行下面的命令来创建 requirements.txt 文件并将 Flask==2.2.2 内容写入该文件。

```
$ echo "Flask==2.2.2" >> requirements.txt
```

熟悉 Python 的同学都知道，有了这两个文件，我们已经可以在本地启动这个 Python Web 应用了，但这不是我们的目标。

接下来，我们开始将这段最简单的 Python 业务代码制作成镜像。

我们需要一个文件来描述镜像是如何被构建的，这个文件叫做 **Dockerfile**，请将以下内容保存为 Dockerfile 文件。

复制代码

```
# syntax=docker/dockerfile:1

FROM python:3.8-slim-buster

RUN apt-get update && apt-get install -y procps vim apache2-utils && rm -rf /va

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .

CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

这个 Dockerfile 只有几行，看起来非常简单，但他代表了一种非常典型的镜像构建的命令。例如 FROM、COPY、RUN、CMD 等命令，它们是从上到下按顺序执行的。当然， Dockefile 还有很多其他命令，你只需要了解最常用的这几个命令就够了。

我们解释一下 Dockerfile 文件里的这几个命令。

第一行以 syntax 开头的是解析器注释，它与 Docker 构建镜像的工具 buildkit 相关，在一般情况，我都建议你使用 docker/dockerfile:1，它代表始终指向最新的语法版本。

FROM 命令，表示使用官方仓库的 python:3.8-slim-buster 镜像作为基础镜像。在我们熟悉的编程方法中，你可以理解为从该镜像继承。这个镜像已经安装了 Python3 和 Pip3 等所有的 Python 相关的工具和包，我们可以直接使用。

RUN 的含义是在镜像内运行指定的命令，这里我们为镜像安装了一些必要的工具。

WORKDIR 的含义是镜像的工作目录，你可以理解为后续所有的命令都将以此为基准路径。这样，我们就可以在后续的命令中使用相对路径而不是完整路径了。

COPY 的含义是将本地的文件或目录复制到镜像内指定的位置。第一个参数代表本地文件或目录，第二个参数代表要复制到镜像内的位置。例如，第七行 COPY 表示，将本地当前目录下的 requirements.txt 文件复制到镜像工作目录 /app 中，文件命名同样为 requirements.txt。

第十行 RUN 的含义是在镜像里运行 pip3 安装 Python 依赖。请注意，这些依赖将会被安装在镜像里而不是本地。

接下来，第十行又出现了一个 COPY 命令，它的含义是将当前目录所有的源代码复制到镜像的工作目录 /app 下，复制目录的语法和我们之前提到的复制文件是类似的。

最后一行 CMD 的含义是镜像的启动命令。在一个 Dockerfile 中，只能有一个 CMD 命令，如果有多个，那么只有最后一个 CMD 命令会起作用。例如，我们希望在镜像被运行时启动 Python Flask Web 服务器，并监听在特定主机上。CMD 的第一个参数 python3 是我们希望运行的可执行命令，后面的参数表示运行 python3 命令所需要的参数。

在一些场景下，你可能会看到另一种与 CMD 类似的命令：ENTRYPOINT。它的功能和 CMD 类似，但又有一些差异。在现阶段，我们只需要先记住 CMD 命令，这可以满足大部分使用场景。

好了，万事俱备，**接下来我们正式开始构建属于自己的第一个镜像。**

我们要在本地电脑的当前目录执行 ls 命令，确认我们刚才保存的 app.py、requirements.txt 以及 Dockefile 是否存在。

复制代码

```
$ ls
Dockerfile      app.py          requirements.txt
```

接下来，**在本机的当前目录下执行 docker build 命令**，这样就可以开始制作镜像了。

复制代码

```
$ docker build -t hello-world-flask .
```

需要注意的是，-t 代表的是我们的镜像名。还记得我们之前提到的镜像版本的概念吗？这里隐含了镜像版本，Docker 会默认用 latest 作为版本号，也就是说，hello-world-flask 与 hello-world-flask:latest 的写法是等价的。

此外，你还需要注意最后面有一个 “.” ，这代表了构建镜像的上下文。现阶段你只需要知道这代表本地源码与执行 docker build 命令的相对位置即可。

执行这条命令时，Docker 会帮我们从官方镜像仓库拉取 python:3.8-slim-buster 镜像，并启动该镜像。接下来，该容器会依次执行我们在 Dockerfile 中书写的命令，例如 WORKDIR、COPY、RUN 等等。

构建好镜像之后，我们可以使用 docker images 命令来查看本地镜像。还记得吗？这条命令我们在之前 docker pull 拉取镜像到本地之后也使用过。

```
$ docker images
REPOSITORY                    TAG         IMAGE ID       CREATED          SIZE
hello-world-flask             latest      3b0803ab8c9c   1 hours ago      121MB
```

接下来我们使用 docker run 启动镜像，验证是不是已经成功把业务代码打包为容器镜像了。

```
$ docker run -d -p 8000:5000 hello-world-flask:latest
```

这里我们再复习一遍 -d 和 -p 参数的含义，你可以花几秒钟的时间回想一下。如果记不清了，记得回去温习一下。

下一步，我们打开浏览器访问 localhost:8000，如果看到以下输出，说明我们已经成功将业务代码打包为了容器镜像。

```
$ Hello, my first docker images!
```

到这里，我们就通过 Python Web 应用的例子，学习了从业务代码打包为容器镜像所涉及到的 Dockerfile 以及相关命令。

构建容器镜像的基本套路

可是，如果你的业务代码是其他编程语言怎么办呢？例如，你用的可能是 Java、Golang、Node.js 等。**这里我为你总结了业务代码构建为容器镜像的基本套路。**

1. 使用 FROM 命令指定一个已经安装了特定编程语言编译工具的基础镜像。你可以在官方镜像仓库找到你所需的任何基础镜像。例如，对于 Java 而言，你可以使用 eclipse-temurin:17-jdk-jammy，对于 Golang 而言，你可以使用 golang:1.16-alpine。
2. 使用 WORKDIR 命令配置一个镜像的工作目录，如 WORKDIR /app。
3. 使用 COPY 命令将本地目录的源码复制到镜像的工作目录下，例如 COPY 。
4. 使用 RUN 命令下载业务依赖，例如 pip3 install。如果是静态语言，那么要进一步编译源码生成可执行文件。
5. 最后，使用 CMD 命令配置镜像的启动命令，也就是将你的业务代码启动起来。

实际上，不管是什么编程语言，构建镜像的方法都是大同小异的。我为你总结的这 5 个方法步骤可以满足我们构建容器镜像的最基本要求。

当然，构建镜像还有很多高级技巧，例如基础镜像的选择、多阶段构建、跨平台构建、buildkit 依赖缓存等等，这些内容在初学过程并不常用，我会在后续的课程里为你详细介绍。

最后，还有一个问题。如果我们想让别人在他的电脑上启动我制作的镜像怎么办？或者，怎么在团队之间共享镜像呢？

还记得我们在前文使用 docker pull 拉取镜像的过程吗？如果我们把自己制作的镜像上传到 Docker 官方的镜像仓库，就可以和其他人共享了。

为了能够上传镜像，你需要先注册一个 Docker Hub 的账号，并且使用 docker login 登录，这和我们使用的 Git 工具类似。

接下来，使用 docke tag 重命名我们之前在本地构建的镜像。

```
$ docker tag hello-world-flask my_dockerhub_name/hello-world-flask
```


请注意，这里需要把 my_dockerhub_name 替换为你实际的 Docker Hub 账户名，也称为镜像仓库的名字。

然后，我们就可以使用 docker push 把本地的镜像上传到 Docker Hub 了。

```
$ docker push my_dockerhub_name/hello-world-flask
```

成功上传后，其他人可以通过 docker pull 命令来拉取我们上传的镜像。

总结

这节课，我们通过一个经典的 Python 应用实例，从 0 开始认识了 Docker 镜像和容器以及 Docker 相关的命令。我还带你了解了如何构建容器镜像，如何上传和分享镜像。

不管使用的是什么编程语言，构建镜像的方法都是差不多的，我为你总结了一套方法论，熟悉了这个方法，工作中要求的 Docker 方面的知识基本上就够用了。

怎么样，现在是不是觉得从 0 开始构建容器镜像也没有这么难？

在下一节课中，我将带你从 0 搭建一个 K8s 集群，并把这节课构建的容器镜像部署到 K8s 集群中。

思考题

最后，给你留一道思考题吧。

在构建镜像的时候，为什么要先 COPY requirements.txt 再安装依赖，而不是 COPY 所有的文件后再安装依赖呢？

欢迎你给我留言交流讨论，你也可以把这节课分享给更多的朋友一起阅读。我们下节课见。

上一篇　开篇词 | 30-60K，转型云原生架构师和SRE需要哪些能力？

下一篇　02 | 如何将容器镜像部署到K8s？

精选留言 (5)

写留言

陈志成

2022-12-12 来自广东

因为 docker 镜像是分层的，先 COPY requirements.txt 再安装依赖，这样 COPY，install 这两层不包含代码文件，这样整体的镜像大小相对更小。

作者回复:　非常正确！

共 3 条评论 >　　6

Daniel
2022-12-12 来自广东

首先感谢老师的文章，比较适合我这种 先感受,再理论（先feel it,再know it)的实战派。

问题中的"所有的文件" 可能是一个动态变化的“一堆文件”，因为后期随着项目的不断迭代，里面会引进一些其它的文件，因此这个所谓的“所有文件”可能是一个动态文件， 而 COPY requirements.txt 在今后的镜像构建里是一个静态的单一文件，因为只有'requirements.txt'。

docker 在构建镜像的时候，dockerfile的会每一个命令会构建一个层，
而在构建的时候是有一个缓存的特点，而这个缓存机制如果是遇到发生变化的层，即使后面的层没有发生变化，也会重新构建，进而并不会用到缓存。
所以，把不变的层放在前面，变的层放在后面，就会让变化之前的层利用到构建镜像的缓存机制，来加速构建镜像的时间。

不知道我说的对不，还请老师指点，我之前有遇到过这个问题。

作者回复: 感谢你对课程的认可，我个人也比较喜欢从实战学习一门技术。

回到你的问题，从缓存的角度上来说是这样的。所以在构建镜像的时候要注意把经常会变化以及不变的区分开，这样可以最大程度利用缓存，加速镜像构建镜像。

2

栖枝
2022-12-14 来自广东

思考题：因为镜像是按照分层构建的，如果每一层的文件没有变化，是不会重复构建的，会使用之前构建好的层，加快构建速度

作者回复:

1

码小呆
2022-12-13 来自广东

因为语句是自上而下执行的,如果都copy文件,那么有些文件的依赖没有安装,那么执行就会保持,镜像就无法构建成功,会失败,所以需要先安装依赖,在copy所有文件

作者回复: 可以实践一下试试看~

Dexter

2022-12-13 来自广东

老师讲一下最新的buildkit和之前docker built之间的异同

作者回复: 对大的区别是 BuildKit 支持并行构建，标准的 docker build 命令是串行构建的。当然它还有其他很多的特性，比如自动垃圾回收、更高效的缓存（支持从远端仓库读取缓存）、不需要 Root 特权等等。

BuildKit 其实不仅仅可以用来构建镜像，最近比较火 Dagger 项目用 BuildKit 做 CI/CD，非常有意思。