BUILDAH: création de meilleures images Docker de build

Table of Contents

Docker est largement reconnu pour fourniture des outils faciles à utiliser pour créer, partager et exécuter des conteneurs, notamment Docker Build Command et Dockerfile.

Ce Lab propose de construire une image de conteneur assez facilement en ce concentrant sur un aspect, la nécessité de créer de petites images, pour deux raisons principales:

- Tout d'abord, si l'image est fréquemment et de manière répétée affichée sur Internet, plus elle sera petite, plus vite cela se produira.
- Deuxièmement, les petits conteneurs contiennent moins de «choses» et donc, plus la surface d'attaque des attaques par les pirates informatiques est petite, plus elles sont dommageables.

  Par conséquent, des conteneurs légers bien conçus ne sont pas seulement bons, car ils chargent plus rapidement Mais ils sont également plus sûrs. Cette article présente des approches pour y parvenir.

Le fichier Docker

À titre d'exemple, un fichier Docker gère tous les packages nécessaires, pour avoir un processus bien défini et reproductible qui crée une image de conteneur , le fichier Dockerfile ressemble à ceci:

```
FROM debian:stretch
MAINTAINER Tim Dudgeon <tdudgeon@informaticsmatters.com>

RUN apt-get update && apt-get install -y \
 build-essential\
 python-numpy\
 cmake\
 python-dev\
```

```
 python-pip\
 sqlite3\
 libsqlite3-dev\
 libboost-dev\
 libboost-system-dev\
 libboost-thread-dev\
 libboost-serialization-dev\
 libboost-python-dev\
 libboost-regex-dev\
 swig\
 git\
 wget\
 zip &&\
 apt-get upgrade -y &&\
 apt-get clean -y

ENV RDKIT_BRANCH=master
RUN git clone -b $RDKIT_BRANCH\
  --single-branch https://github.com/rdkit/rdkit.git

ENV RDBASE=/rdkit
ENV LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$RDBASE/lib:/usr/lib/x86_64-linux-gnu
ENV PYTHONPATH=$PYTHONPATH:$RDBASE

RUN mkdir $RDBASE/build
WORKDIR $RDBASE/build

RUN cmake -DRDK_BUILD_INCHI_SUPPORT=ON .. &&\
 make &&\
 make install &&\
 make clean

WORKDIR $RDBASE
```

L'approche est raisonnablement claire:

- Installe toutes les dépendances à l’aide du gestionnaire de paquets apt (nous utilisons une image de base Debian).
- Clone le référentiel RDKit Git avec le code source.
- Génére l'application RDKit, puis l'installe.

La construction prend environ une heure, mais on obtient finalement une image qui peut être utilisée pour exécuter RDKit:

```
$ docker build -f Dockerfile-all-in-one .
...
... lots and lots of output ...
...
Successfully built bae5c2ce64a8
$ docker run -it --rm bae5c2ce64a8 python
```

```
Python 2.7.13 (default, Nov 24 2017, 17:33:09)
[GCC 6.3.0 20170516] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import rdkit
>>>
```

Inconvénients de la méthode

Bien qu'il s'agisse d'un bon moyen d'illustrer et de reproduire avec certitude le processus de construction d'une application, il présente un certain nombre de problèmes importants.

- Cela prend beaucoup de temps à construire.
- Cette longue durée de construction empêche l'utilisation de générations automatisées sur DockerHub car la génération a expiré.
- L'image obtenue a une taille de 1,25 Go.
- L'image résultante contient un grand nombre de packages et de fichiers présents, mais non nécessaires pour exécuter RDKit.

C'est le cas extrême d'un anti-motif méchant qui affecte presque toutes les images de conteneur que l'on trouve sur DockerHub ou d'autres référentiels, c'est-à-dire que l'image résultante contient divers artefacts nécessaires. Pour construire l'image, mais pas nécessaire d'exécuter dans le conteneur une fois qu'il est construit.

En l'occurence, le conteneur contient git, wget et l'ensemble de l'infrastructure de construction, y compris make, cmake, gcc et g ++, ainsi que le gestionnaire de paquets apt, ainsi que le référentiel RDKit GitHub extrait. Alors que RDKit, qui est l'unique objectif de cette image de conteneur.

Donc, ce fichier Dockerfile est utile pour illustrer comment construire différentes versions de RDKit, et pourrait même être utile pour un développeur RDKit qui a besoin de reconstruire des choses et de faire du piratage, c'est un piètre exemple de la façon de créer un conteneur pour simplement exécuter RDKit et bien d'autres choses nécessaires pour construire ce code source. C'est énorme et a une assez grande surface d'attaque avec tous ces extras inutiles.

De toute évidence, il y a un meilleur moyen de le faire.

Méthode 1: Utilisation d'un conteneur de build

L'installation des packages RPM et DEB résoudrait certainement une grande partie du problème, mais cette approche ne permet pas de conserver les packages RPM et DEB à jour, et cela ne permet pas d'avoir la souplesse nécessaire pour utiliser les versions les plus récentes qui ne seraient pas disbonibles dans les distributions standard.

Il est préférable de construire les paquets, puis de les installer directement sans avoir besoin d'un gestionnaire de paquets.

Il faut donc utiliser un modèle de construction, qui consiste à utiliser une grande image pour construire les artefacts, puis à installer ces packages dans une deuxième image légère, orchestrée par un simple script bash.

Docker a récemment introduit des versions en plusieurs étapes qui peuvent également être utilisées pour obtenir à peu près la même chose, mais il s'agit d'une fonctionnalité relativement nouvelle qui n'a pas encore été intégrée à certaines distributions Linux.

Etape 1 : Conteneur de build

L'étape 1 consiste à utiliser un fichier Docker similaire à celui décrit dans l'article précédent, qui construit RDKit à partir du code source, mais plutôt que d'installer la version résultante dans l'image, nous construisons des packages DEB et RPM. Le fichier Docker est celui-ci

```
# Dockerfile for building RDKit artifacts.
# This is a heavyweight image containing all aspects of RDKit plus the build
system.
# It's purpose is to create the RDKit artifacts that will be deployed to
lighter weight images.

# Latest RDKit now needs cmake 3.1 which is not preset on jessie so we must
use buster
FROM debian:buster
LABEL maintainer="Tim Dudgeon<tdudgeon@informaticsmatters.com>"

ARG GIT_REPO
ARG GIT_BRANCH=master
ARG GIT_TAG
ARG POSTGRES_VERSION=11

# This adds the postgres apt repos as postgresql-10 is not available for
buster
# and postgresql-11 does not seem to work with RDKit yet.
#
RUN apt-get update &&\
 apt-get -y install curl ca-certificates gnupg &&\
 curl https://www.postgresql.org/media/keys/ACCC4CF8.asc | apt-key add - &&\
 echo "deb http://apt.postgresql.org/pub/repos/apt/ buster-pgdg main" >
/etc/apt/sources.list.d/pgdg.list

RUN apt-get update &&\
  #apt-get upgrade -y &&\
  apt-get install -y --no-install-recommends \
  build-essential\
  python3-dev\
```

```
  python3-numpy\
  python3-pip\
  cmake\
  sqlite3\
  libsqlite3-dev\
  libboost-dev\
  libboost-system1.67-dev\
  libboost-thread1.67-dev\
  libboost-serialization1.67-dev\
  libboost-python1.67-dev\
  libboost-regex1.67-dev\
  libboost-iostreams1.67-dev\
  zlib1g-dev\
  swig\
  libeigen3-dev\
  git\
  wget\
  openjdk-11-jdk\
  postgresql-$POSTGRES_VERSION\
  postgresql-server-dev-$POSTGRES_VERSION\
  postgresql-plpython3-$POSTGRES_VERSION\
  zip\
  unzip &&\
  apt-get clean -y


RUN if [ $GIT_TAG ]; then echo "Checking out tag $GIT_TAG from repo
$GIT_REPO branch $GIT_BRANCH"; else echo "Checking out repo $GIT_REPO branch
$GIT_BRANCH"; fi
RUN git clone -b $GIT_BRANCH --single-branch $GIT_REPO &&\
  if [ $GIT_TAG ]; then cd rdkit && git fetch --tags && git checkout
$GIT_TAG; fi

# hack to build cartridge packages. can be removed once this code hits the
repo
COPY patch_pgsql_rpm.patch /rdkit
RUN cd /rdkit && patch -p1 < patch_pgsql_rpm.patch

ENV RDBASE=/rdkit
ENV
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$RDBASE/lib:$RDBASE/Code/JavaWrappers/gmwra
pper:/usr/lib/x86_64-linux-gnu
ENV PYTHONPATH=$PYTHONPATH:$RDBASE
ENV JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
ENV CLASSPATH=$RDBASE/Code/JavaWrappers/gmwrapper/org.RDKit.jar

RUN mkdir $RDBASE/build
WORKDIR $RDBASE/build

RUN cmake -Wno-dev\
  -DPYTHON_EXECUTABLE=/usr/bin/python3\
```

```
  -DRDK_INSTALL_INTREE=OFF\
  -DRDK_BUILD_INCHI_SUPPORT=ON\
  -DRDK_BUILD_AVALON_SUPPORT=ON\
  -DRDK_BUILD_PYTHON_WRAPPERS=ON\
  -DRDK_BUILD_SWIG_WRAPPERS=ON\
  -DRDK_BUILD_PGSQL=ON\
  -DPostgreSQL_ROOT=/usr/lib/postgresql/$POSTGRES_VERSION\
  -
DPostgreSQL_TYPE_INCLUDE_DIR=/usr/include/postgresql/$POSTGRES_VERSION/serve
r\
  -DCMAKE_INSTALL_PREFIX=/usr\
  -DCPACK_PACKAGE_RELOCATABLE=OFF\
  ..

RUN nproc=$(getconf _NPROCESSORS_ONLN)\
  && make -j $(( nproc > 2 ? nproc - 2 : 1 ))
RUN make install
RUN sh Code/PgSQL/rdkit/pgsql_install.sh
RUN cpack -G DEB

WORKDIR $RDBASE

,, mais l'élément clé est Ce:

RUN cmake -Wno-dev \
  -DRDK_INSTALL_INTREE=OFF \
  -DRDK_BUILD_INCHI_SUPPORT=ON \
  -DRDK_BUILD_AVALON_SUPPORT=ON \
  -DRDK_BUILD_PYTHON_WRAPPERS=ON \
  -DRDK_BUILD_SWIG_WRAPPERS=ON \
  -DCMAKE_INSTALL_PREFIX=/usr \
  ..

RUN nproc=$(getconf _NPROCESSORS_ONLN)\
  && make -j $(( nproc > 2 ? nproc - 2 : 1 ))\
  && make install\
  && cpack -G DEB\
  && cpack -G RPM
```

Le script bash principal fonctionne comme ceci:

```
docker build -f Dockerfile-build-debian\
  -t $BASE/rdkit-build:$TAG\
  --build-arg RDKIT_BRANCH=$BRANCH .
```

RDKit est construit comme auparavant, mais `cpack` est ensuite utilisé pour créer les packages RPM et DEB. On peut également créer les deux, ici sur un système Debian. On construit également les artefacts nécessaires à une image RDKit basée sur Java.

Etape 2: Conteneurs finaux

La deuxième étape crée un conteneur en cours d'exécution à partir de cette image et copie les artefacts créés du conteneur sur le système hôte.

```
rm -rf artifacts/$TAG
mkdir -p artifacts/$TAG
mkdir artifacts/$TAG/debs
mkdir artifacts/$TAG/rpms
mkdir artifacts/$TAG/java
docker run -it --rm -u $(id -u)\
  -v $PWD/artifacts/$TAG:/tohere:Z\
  $BASE/rdkit-build:$TAG bash -c\
  'cp build/*.deb /tohere/debs && cp build/*.rpm /tohere/rpms && cp
Code/JavaWrappers/gmwrapper/org.RDKit.jar /tohere/java && cp
Code/JavaWrappers/gmwrapper/libGraphMolWrap.so /tohere/java'
```

Les étapes permettent de créer différentes images Docker à des fins différentes.

- Une image basée sur Debian avec Python utilisant les packages DEB (Dockerfile-python-debian).
- Une image Centos7 avec Python utilisant les packages RPM (Dockerfile-python-centos).
- Une image basée sur Debian avec Java utilisant les paquets DEB (Dockerfile-java-debian).
- Une image basée sur Debian avec Java et le conteneur de servlets Tomcat utilisant les packages DEB (Dockerfile-tomcat-debian).

Mais quelle est la taille de ces images et comment se compare-t-elle à l'approche précédente qui donnait une image de 1,25 Go?

```
$ docker images
REPOSITORY                                          TAG
IMAGE ID            CREATED             SIZE
informaticsmatters/rdkit-tomcat-debian              latest
7fa32622d1fe        31 hours ago        381 MB
informaticsmatters/rdkit-java-debian                latest
60c9fc7b7c72        31 hours ago        357 MB
informaticsmatters/rdkit-python-centos              latest
380a50f7ddd3        31 hours ago        542 MB
informaticsmatters/rdkit-python-debian              latest
eacb6065c14c        31 hours ago        414 MB
informaticsmatters/rdkit-build                      latest
7b2cc073b265        31 hours ago        2.27 GB
```

On peut voir que l'image de compilation (rdkit-build) est encore plus grande à 2,27 Go, mais elle est attendue car elle contient également les packages RPM et DEB. La vraie comparaison est l'image rdkit-python-debian qui s'élève à 414Mo. , 33% de la taille de l'original, c'est une amélioration considérable!

l'image basée sur les centos est un peu plus grande, à 542 Mo, car l'image debian:jessie sur la version de Docker Hub est de 100 Mo, alorsque l'image centos:7 correspond à 204 Mo.

On a donc réussi à créer une image de conteneur beaucoup plus petite, plus efficace et plus sûre, mais ces images ne sont pas encore idéales pour plusieurs raisons. Les paquets DEB ou RPM sont bloqués dans les images résultantes, ce qui absorbe des octets inutiles et, il y as encore des paquets inutiles installés - ceux des gestionnaires de paquets eux-mêmes. On va donc modifier un peu le processus de construction pour résoudre le premier problème, Mais pour la seconde, il faudra appliquer une approche plus radicale.

Méthode 2: Utilisation de buildah

Cette section présente l'utilisation de buildah pour générer des images de conteneur ne contenant que le nécessaire, sans aucun contenu supplémentaire. On montrera comment cela peut permettre de générer de très petites images qui se chargeront plus rapidement et seront plus sécurisées, et ce, sans avoir besoin du démon Docker.

Presque toutes les images de conteneur sont des images Docker construites à partir d'un fichier Docker à l'aide de la génération de dockers. Mais ce n'est pas le c'est le seul moyen de le faire. Ce n'est pas le cas. Cet article porte sur un outil relativement nouveau appelé buildah, qui présente certains avantages clés par rapport à l'utilisation des outils Docker:

- Aucun processus démon en cours d'exécution (le démon Docker)
- Emballe l'image de l'extérieur et non de l'intérieur

De nombreuses images utilisent Centos7 et Debain comme images de base:

```
docker images

REPOSITORY              TAG           IMAGE ID            CREATED
SIZE
docker.io/centos        7             3fa822599e10        6 months ago
204 MB
docker.io/debian        buster        ebdc13caae1e        2 months ago
106 MB
```

On remarquera que l'image Centos a une taille presque deux fois supérieure à celle de l'image Debian.

C'est en partie parce que Centos utilise Yum en tant que gestionnaire de paquets et que celui-ci est basé sur Python, de sorte que l'image Centos est livrée avec une installation complète de Python et que Python n'est guère léger! Contrairement au gestionnaire de paquets apt de Debian. Pas besoin de Python.

Cela cache un autre problème avec ces deux images: elles contiennent un gestionnaire de paquets nécessaire en raison du fonctionnement du processus de génération de docker avec son fichier Docker. En général, la première chose à faire dans un fichier Dockerfile consiste à installer les paquets dont on a besoin de yum or même si le gestionnaire de paquets ne sert plus après l'installation, il fera partie de l'image résultante, ce qui signifie que l'image finale a une taille supplémentaire et un vecteur d'attaque plus grand pour les pirates. Par exemple, si on veut une image qui exécute **nginx** et que cette image ne contienne uniquement que **nginx**, mais pas tout ce qui était nécessaire pour y placer **nginx**.

C'est ici que buildah est différent: il installe les paquetages de l'extérieur et l'image résultante n'inclut pas le gestionnaire de paquets ni tout ce qui était nécessaire pour construire ou installer les outils.

Etape 1: Conteneur de Build

Pour les images basées sur Centos, il faut créer un environnement dans une image Centos Docker. Lancer le conteneur, mettre à jour les packages et installer Buildah:

```
docker run -it -v $PWD:$PWD:Z -v /var/lib/containers:/var/lib/containers --
privileged -w $PWD centos:7 bash
[root@441275f3a875 buildah]# yum update -y
...
[root@441275f3a875 buildah]# yum install -y buildah
...
[root@441275f3a875 buildah]#
```

Dans ce contexte, les fichiers Docker peuvent toujours fonctionner avec Buildah: comme exemple, on peut utiliser ce fichier Dockerfile simple pour créer un conteneur nginx:

```
FROM centos:7

RUN yum install -y epel-release &&\
  yum update -y &&\
  yum -y install nginx --setopt install_weak_deps=false &&\
  yum -y clean all

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Maintenant construisons ceci avec buildah

```
buildah bud .
...
...
STEP 3: EXPOSE 80
STEP 4: CMD ["nginx", "-g", "daemon off;"]
STEP 5: COMMIT containers-
```

```
storage:[overlay@/var/lib/containers/storage+/var/run/containers/storage:ove
rlay.override_kernel_check=true]@7540f22d0d568655fb1d02a20b250911cdf1564b6fe
84754f83528df6f082da1
Getting image source signatures
Skipping fetch of repeat blob
sha256:43e653f84b79ba52711b0f726ff5a7fd1162ae9df4be76ca1de8370b8bbf9bb0
Copying blob
sha256:0f499735cf675b07294a2add64622d105769a4aab08c96f5f63a886225d59088
 74.35 MiB / 74.35 MiB
[======================================================] 2s
Copying config
sha256:40f2bd546f05497db4ce9544b5072064421a1f192cbd2df97a3a77c142a44029
 1.17 KiB / 1.17 KiB
[======================================================] 0s
Writing manifest to image destination
Storing signatures
7540f22d0d568655fb1d02a20b250911cdf1564b6fe84754f83528df6f082da1
[root@417a3c92073d buildah]# buildah images
IMAGE ID             IMAGE NAME                      CREATED AT
SIZE
7540f22d0d56         <none>                          May 31, 2018 09:41
419.3 MB
```

On a donc créé une image à partir du fichier Docker, dans lee conteneur Centos où on travaille, le démon Docker est en cours d'exécution. On a créé une image Docker sans Docker.

Etape 2: Conteneurs finaux

Passons maintenant à l'aspect le plus intéressant de Buildah, la capacité d'emballer des images de l'extérieur, en utilisant un gestionnaire de paquets pour installer des paquets dans un fichier Docker. Une ligne comme celle-ci installe NGinx:

```
RUN yum -y install nginx
```

Le problème avec ceci, comme indiqué précédemment, est que le gestionnaire de packages (et dans ce cas également Python) fait partie de l'image générée, mais ni yum ni Python ne sont nécessaires pour exécuter Nginx. Avec Builah, on utilise toujours un Package Manager, mais il s'exécute sur la machine hôte et installe les packages dans le système de fichiers qui deviendra l'image à créer. Le gestionnaire de packages ne fait pas partie de cette image.

Donc, avec buildah, on va créer une image de base Centos ne contenant ni yum ni Python, et nous obtiendrons une petite image.

Le script de construction ressemble à ceci:

```
#!/bin/bash
```

```
set -x

# construire une image minimale
newcontainer=$(buildah from scratch)
scratchmnt=$(buildah mount $newcontainer)

# installe les paquets
yum install bash coreutils --installroot $scratchmnt --releasever 7\
  --setopt install_weak_deps=false --setopt=tsflags=nodocs\
  --setopt=override_install_langs=en_US.utf8 -y
yum clean all -y --installroot $scratchmnt --releasever 7
rm -rf $scratchmnt/var/cache/yum

# définit des informations de configuration
buildah config --label name=centos-base $newcontainer

# commit l'image
buildah unmount $newcontainer
buildah commit $newcontainer tdudgeon/centos-base
```

Dans ce cas, on utilise buildah, non pas à partir d'un fichier Dockerfile, mais sous la forme d'un ensemble de commandes exécuté en tant que script bash. On crée une nouvelle image minimale, contenant simplement le kenrnel Linux à partir de la machine hôte. Puis on installe bash et coreutils à l'aide du gestionnaire de paquets yum, mais celui-ci est celui de la machine hôte et installe ces paquets dans le système de fichiers monté en tant que scratchmnt. Le paquet coreutils installé contient un ensemble Parmi les outils standard de Linux, à vrai dire, nombre d'entre eux ne sont pas nécessaires, mais sans eux, on aura du mal à déboguer le conteneur. Si on a besoin de lancer un shell, on peut également installer le sous-ensemble de ceux-ci.

```
buildah images

IMAGE ID             IMAGE NAME
CREATED AT             SIZE
06c90c079f3a         docker.io/tdudgeon/centos-base:latest
May 31, 2018 10:48     56.96 MB
```

La taille finale des images est d'environ 57 Mo, alors que l'équivalent de Docker Hub était de 204 Mo et l'image Debian de 106 Mo.

Si on inclue également yum dans les paquetages installés, la taille de l'image passe à 120 Mo. La taille supplémentaire de l'image Centos de Docker Hub par rapport à celle de Debian est due à la présence de yum et de Python.

Pour pouvoir utiliser cette petite image dans Docker il faut copier les images de /var/lib/containers vers où Docker l'attend. Pour cela, on a besoin du démon Docker. En dehors de l'image Docker procéder ainsi:

```
sudo buildah push tdudgeon/centos-base docker-daemon:tdudgeon/centos-base:latest
...
docker run -it --rm tdudgeon/centos-base bash
bash-4.2#
```

On a maintenant une image de base Centos qui représente presque le quart de celle de DockerHub.