Docker: Créer une image de base

Table of Contents

La plupart des fichiers `Dockerfiles` partent d'une image parent. Lorsqu'on souhaite contrôler complètement le contenu de l'image, on peut créer une image de base. Voici la différence:

- Une image parent est l'image sur laquelle l'image est basée. Elle fait référence au contenu de la directive `FROM` dans le fichier `Dockerfile`. Chaque déclaration ultérieure dans le fichier `Dockerfile` modifie cette image parente.
- Une image de base n'a pas de ligne `FROM` dans son fichier `Dockerfile`.

Cette article présente plusieurs façons de créer une image de base. Le processus spécifique dépendra fortement de la distribution Linux que l'on souhaite empaqueter.

Choix de l'OS

Pour aider à faire un choix qui correspond aux besoins, cette section passe en revue certains des critères pertinents afin de l'éclairer et présente le choix d'un OS sous l'angle de la taille sur le disque de l'image.

Crières communs pour choisir une image de base

Il existe un certain nombre de critères communs pour choisir une image de base:

- **Stabilité**: lorsqu'on souhaite qu'une version compilée aujourd'hui fournisse le même ensemble de base de bibliothèques, la même structure de répertoires et l'infrastructure que la version compilée de demain. Le besoin de stabilité suggère de ne pas utiliser de systèmes d'exploitation à durée de vie limitée, comme Fedora ou les versions non-LTS Ubuntu.
- **Mises à jour de sécurité**: lorsqu'on souhaite que l'image de base soit bien conservée, afin que les mises à jour de sécurité du système d'exploitation de base soient facilitées dans les meilleurs délais.
- **Dépendances à jour**: à moins de créer une application très simple, celle-ci dépendra probablement des bibliothèques et des applications installées sur le système d’exploitation (par exemple, un compilateur). Il est préférable qu’elles ne soient pas trop anciennes.
- **Dépendances étendues**: pour certaines applications, des dépendances moins courantes peuvent être requises: une image de base avec accès à un grand nombre de bibliothèques facilite cette opération.
- **Taille de l'images**: toutes choses étant égales par ailleurs, il est préférable d’avoir une image Docker plus petite qu’une image Docker plus grande.

Choix d'une image sur le critère de taille

En plus des avantages des de taille évidente qu'offrent les images plus petites, l'environnement devrait être également plus facile çà maintenir et plus efficace: les petites images augmentent la sécurité tout en réduisant la taille de l'empreinte de sécurité.

Afin de produire un image docker la plus petite possible elle doit contenir uniquement les éléments de base nécessaires à l'exécution de l'application.

La liste ci dessous compile les images les plus populaires du système d'exploitation de base, basées sur la taille du fichier ainsi que quelques critères qualitatifs:

- **BusyBox**: 2 Mo de taille - Busybox a été écrit dans un souci d'optimisation de la taille et de ressources limitées. Il est également extrêmement modulaire, permettant d'inclure ou d'exclure facilement des commandes (ou fonctionnalités) au moment de la compilation.
- **Alpine**: 5 Mo de taille - Une image Busybox plus complète avec accès à un référentiel de packages
- **Cirros**: 8 Mo de taille - Il s’agit d’un petit système d’exploitation spécialisé dans le cloud
- **Debian**: 125 Mo de taille - Utilise le noyau Linux et les outils de base du projet GNU
- **CentOS**: 172 Mo de taille - dérivés de Red Hat Linux Enterprise RHEL et chaque version est

prise en charge pendant 10 ans
- **Fedora**: 187 Mo de taille - Commandité par Red Hat Linux Enterprise et s'efforce de stimuler les innovations
- **Ubuntu**: 188 Mo de taille - L'image la plus populaire
- **Windows Nano Server**: 300 Mo de taille - ne contient qu'un sous-ensemble minimal de bibliothèques Windows principales et uniquement en mode 64 bits, peut être utilisé pour les logiciels natifs cloud / conteneur qui n'utilisent que des bibliothèques d'exécution 64 bits prises en charge ou une application construite sur .NET Core (pour les applications en console) ou ASP.NET Core (pour les applications Web).
- **Windows Server Core**: 3-4GB de taille - peut être utilisée pour les logiciels destinés à être exécutés dans un environnement Windows pris en charge, une prise en charge MSI, une prise en charge complète de .NET Framework, une exécution 32 bits et 64 bits, etc.

Créer une image complète en utilisant tar

En général, il faut commencer par une machine exécutant la distribution que l'on souhaite empaqueter en tant qu'image parent, bien que cela ne soit pas nécessaire pour certains outils tels que `Debian Debootstrap`, que l'on peut également utiliser pour créer des images Ubuntu.

Cela peut être aussi simple que cela pour créer une image parent Ubuntu:

```
$ sudo debootstrap xenial xenial > /dev/null
$ sudo tar -C xenial -c . | docker import - xenial
```

```
$ docker run xenial cat /etc/lsb-release

DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=16.04
DISTRIB_CODENAME=xenial
DISTRIB_DESCRIPTION="Ubuntu 16.04 LTS"
```

Construction d'une image de base Arch Linux

Cette section détaille comment créer une image `Arch Linux` à partir de zéro ainsi que les bonnes pratiques pour créer une image afin de connaître exactement ce qui est installé dans le conteneur.

Comme point de départ, on utilise le script `mkimage-arch.sh` utilisé dans le lab:

Préparion de l'nvironnement de construction

Force l'exécution en tant que ROOT

Le script doit être exécuté en tant que root

```
if [ “$(id -u)” != “0” ]; then
printf "This script must be run as root\n"
exit 1
fi
```

Vérification des prérequis

Il y a deux paquets requis: `expect` + `pacstrap` qui peut être installé avec:

```
$ pacman -S arch-install-scripts expect
```

Le script vérifie ceci comme suit:

```
hash pacstrap &>/dev/null || {
    printf "Could not find pacstrap. Run pacman -S arch-install-scripts"
    exit 1
}

hash expect &>/dev/null || {
    printf "Could not find expect. Run pacman -S expect"
    exit 1
}
```

Le || est similaire à && sauf qu'il indique au shell d'évaluer uniquement l'expression après, lorsque la première expression échoue.

La commande hash affecte la manière dont l'environnement shell actuel se souvient des emplacements des utilitaires trouvés. Si exécuté sans aucun paramètre, il indique le chemin de toutes les commandes exécutées depuis la dernière réinitialisation du hachage (hash -r), par exemple.

```
$ hash
hits    command
   1    /usr/bin/git
   1    /usr/bin/vim
   3    /usr/bin/cat
   1    /usr/bin/touch
   1    /usr/bin/mv
   1    /usr/bin/mkdir
   3    /usr/bin/man
  13    /usr/bin/ls
```

La table de hachage est une fonctionnalité de bash qui l’empêche de chercher dans $PATH chaque fois qu'on tape une commande en mettant en cache les résultats en mémoire. Pour ce cas

d'utilisation, on l'utilise comme moyen de tester si la commande est disponible.

Définition des variables de base pour créer une image

```
PACMAN_EXTRA_PKGS=''
EXPECT_TIMEOUT=60
ARCH_KEYRING=archlinux
DOCKER_IMAGE_NAME=archlinux
```

Création du dossier d'installation

On créé un système de fichiers racine temporaire avec mktemp:

```
ROOTFS=$(mktemp -d /tmp/rootfs-archlinux-XXXXXXXXXX)
```

`mktemp` crée un répertoire temporaire (ou un fichier) basé sur le modèle fourni pour randomiser le nom. Chaque valeur X est remplacée par une chaîne aléatoire.

puis on définit les autorisations

```
chmod 755 "$ROOTFS"
```

Préparion de la configuration de pacman

Définition des packages à ne pas installer

Afin de produire une image minimale, on définit une liste des packages à ne pas installer:

```
PKGIGNORE=(
    cryptsetup
    device-mapper
    dhcpcd
    iproute2
    jfsutils
    linux
    lvm2
    man-db
    man-pages
    mdadm
    nano
    netctl
    openresolv
    pciutils
```

```
    pcmciautils
    reiserfsprogs
    s-nail
    systemd-sysvcompat
    usbutils
    vi
    xfsprogs
)
```

Puis on développe un tableau sans index ne donne que le premier élément.

$IFS est une variable de shell spéciale qui spécifie le séparateur interne de champ .

```
IFS=','
PKGIGNORE="${PKGIGNORE[*]}"
unset IFS
printf "%s""\nPackages not to be installed : $PKGIGNORE\n"
```

Définition de pacman.conf

On définit la variable PACMAN_CONF avec le fichier de configuration créé précédemment

```
PACMAN_CONF='./arch-docker-pacman.conf'
```

Définit le miroir pour pacman

On définit la variable PACMAN_MIRRORLIST en fournissant l'url des dépôts AUR

```
PACMAN_MIRRORLIST='Server =
https://mirrors.kernel.org/archlinux/\$repo/os/\$arch'
export PACMAN_MIRRORLIST
```

Installation du système

Définition de la lanque

```
export LANG="C.UTF-8"
```

Définition du fuseau horaire

```
arch-chroot "$ROOTFS" /bin/sh -c "ln -s /usr/share/zoneinfo/UTC
/etc/localtime"
```

Définition des paramètres régionaux

```
echo 'en_US.UTF-8 UTF-8' > "$ROOTFS"/etc/locale.gen
arch-chroot "$ROOTFS" locale-gen
```

Exécution de pacman

Pour une installation unattented On utilise `expect` pour répondre automatiquement à `pacstrap`

`Expect` est un programme qui “parle” à d'autres programmes interactifs selon un script. Dans l'exemple du script ci-dessous:

- **spawn** lance le processus `pacstrap` avec les arguments `-C $PACMAN_CONF -c -d -G -i $ROOTFS base haveged $PACMAN_EXTRA_PKGS --ignore $PKGIGNORE`
- **expect** attend que le précédent processus génère une des chaînes “prévues” (`"anyway? [Y/n] "`, `"(default=all): "` ou `"installation? [Y/n]"` pour répondre avec `send`
- **send** répond au processus engendré `stdin`

```
expect <<EOF
    set send_slow {1 .1}
    proc send {ignore arg} {
        sleep .1
        exp_send -s -- \$arg
    }
    set timeout $EXPECT_TIMEOUT

     spawn pacstrap -C $PACMAN_CONF -c -d -G -i $ROOTFS base haveged
$PACMAN_EXTRA_PKGS --ignore $PKGIGNORE
    expect {
        -exact "anyway? \[Y/n\] " { send -- "n\r"; exp_continue }
        -exact "(default=all): " { send -- "\r"; exp_continue }
        -exact "installation? \[Y/n\]" { send -- "y\r"; exp_continue }
    }
EOF
```

Suppresion des fichiers manuels pour économiser de l'espace

```
arch-chroot "$ROOTFS" /bin/sh -c 'rm -r /usr/share/man/*'
```

Rechargement du trousseau de clés de Pacman

On utilise haveged pour générer des nombres aléatoires et alimenter un périphérique aléatoire Linux. On peut exécuter pacman-key -init avant d'utiliser pacman pour la première fois; le trousseau de clés local peut ensuite être rempli avec les clés de tous les emballeurs officiels d'Arch Linux avec pacman-key –populate archlinux.

```
arch-chroot "$ROOTFS" /bin/sh -c "haveged -w 1024; pacman-key --init; pkill
haveged; pacman -Rs --noconfirm haveged; pacman-key --populate
$ARCH_KEYRING; pkill gpg-agent"
```

Puis on charge pacman mirrorlist avec le contenu de PACMAN_MIRRORLIST

```
arch-chroot "$ROOTFS" /bin/sh -c "echo $PACMAN_MIRRORLIST >
/etc/pacman.d/mirrorlist"
```

Reconstruction de /dev

udev est un gestionnaire de périphériques pour le noyau Linux. Il gère principalement les nœuds de périphériques dans le répertoire /dev et gère également tous les événements d'espace utilisateur générés lorsque des périphériques sont ajoutés au système ou supprimés de celui-ci, y compris le chargement du microprogramme requis par certains périphériques

udev ne fonctionne pas dans les conteneurs, il faut donc reconstruire manuellement /dev

```
DEV=$ROOTFS/dev
rm -rf "$DEV"
mkdir -p "$DEV"
mknod -m 666 "$DEV"/null c 1 3
mknod -m 666 "$DEV"/zero c 1 5
mknod -m 666 "$DEV"/random c 1 8
mknod -m 666 "$DEV"/urandom c 1 9
mkdir -m 755 "$DEV"/pts
mkdir -m 1777 "$DEV"/shm
mknod -m 666 "$DEV"/tty c 5 0
mknod -m 600 "$DEV"/console c 5 1
mknod -m 666 "$DEV"/tty0 c 4 0
mknod -m 666 "$DEV"/full c 1 7
mknod -m 600 "$DEV"/initctl p
mknod -m 666 "$DEV"/ptmx c 5 2
ln -sf /proc/self/fd "$DEV"/fd
```

Construction et test de l'image

Construction de l'image tar

Chargement du système de fichiers racine dans un fichier tar et import de l' image dans Docker.

```
tar --numeric-owner --xattrs --acls -C "$ROOTFS" -c . | docker import -
"$DOCKER_IMAGE_NAME"
```

Options tar utilisées:

- `--numeric-owner` Toujours utiliser des nombres pour les noms d'utilisateur/groupe.
- `--xattrs` Activer le support des attributs étendus
- `--acls` Activer la prise en charge des ACL POSIX
- `-C` Changer de répertoire
- `-c` Créer une nouvelle archive

Test de la nouvelle image

```
docker run --rm -t $DOCKER_IMAGE_NAME echo Success
```

L'exécution du script devrait se terminer par

```
tar: ./etc/pacman.d/gnupg/S.gpg-agent: socket ignored
sha256:82b0356924efb95e5483417dd4f0cef85fce7afbacb75c18632de7bc45edd796
Success
```

Suppression du système de fichiers racine temporaire

```
rm -rf "$ROOTFS"
```