EBook Gratuito

APPRENDIMENTO

Docker

Free unaffiliated eBook created from
Stack Overflow contributors.

#docker

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: docker

It is an unofficial and free Docker ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Docker.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con Docker

Osservazioni

Docker è un progetto open-source che automatizza la distribuzione di applicazioni all'interno di contenitori software. Questi contenitori di applicazioni sono simili a macchine virtuali leggere, in quanto possono essere eseguiti separatamente l'uno dall'altro e l'host in esecuzione.

Docker richiede che le funzionalità presenti negli ultimi kernel di Linux funzionino correttamente, quindi su host Mac OSX e Windows è necessaria una macchina virtuale su cui è in esecuzione Linux per il corretto funzionamento della finestra mobile. Attualmente il metodo principale di installazione e configurazione di questa macchina virtuale è tramite Docker Toolbox che utilizza VirtualBox internamente, ma è prevista l'integrazione di questa funzionalità nella finestra mobile stessa, utilizzando le funzionalità di virtualizzazione native del sistema operativo. Sui sistemi Linux la finestra mobile viene eseguita in modo nativo sull'host stesso.

Versioni

Versione	Data di rilascio
17.05.0	2017/05/04
17.04.0	2017/04/05
17.03.0	2017/03/01
1.13.1	2016/02/08
1.12.0	2016/07/28
1.11.2	2016/04/13
1.10.3	2016/02/04
1.9.1	2015/11/03
1.8.3	2015/08/11
1.7.1	2015/06/16
1.6.2	2015/04/07
1.5.0	2015/02/10

Examples

Installazione di Docker su Mac OS X

Requisiti: OS X 10.8 "Mountain Lion" o più recente richiesto per eseguire Docker.

Mentre il binario docker può essere eseguito in modo nativo su Mac OS X, per costruire e ospitare i contenitori è necessario eseguire una macchina virtuale Linux sulla scatola.

1.12.0

Dalla versione 1.12 non è necessario avere una VM separata da installare, poiché Docker può utilizzare la funzionalità nativa `Hypervisor.framework` di OSX per avviare una piccola macchina Linux che funga da backend.

Per installare la finestra mobile seguire i seguenti passi:

1. Vai a Docker per Mac
2. Scarica ed esegui il programma di installazione.
3. Continua attraverso il programma di installazione con le opzioni predefinite e inserisci le credenziali del tuo account quando richiesto.

Controlla qui per ulteriori informazioni sull'installazione.

1.11.2

Fino alla versione 1.11 il modo migliore per eseguire questa VM Linux è installare Docker Toolbox, che installa Docker, VirtualBox e la macchina guest Linux.

Per installare la casella degli strumenti finestra mobile, attenersi alla seguente procedura:

1. Vai a Docker Toolbox
2. Fare clic sul collegamento per Mac ed eseguire il programma di installazione.
3. Continua attraverso il programma di installazione con le opzioni predefinite e inserisci le credenziali del tuo account quando richiesto.

Questo installerà i binari Docker in `/usr/local/bin` e aggiornerà qualsiasi installazione di Virtual Box esistente. Controlla qui per ulteriori informazioni sull'installazione.

Per verificare l'installazione:

1.12.0

1. Avvia `Docker.app` dalla cartella Applicazioni e assicurati che sia in esecuzione. Quindi apri Terminale.

1.11.2

1. Apri il `Docker Quickstart Terminal` , che aprirà un terminale e lo preparerà per l'uso per i comandi Docker.

2. Una volta che il terminale è di tipo aperto

```
$ docker run hello-world
```

3. Se tutto va bene, questo dovrebbe stampare un messaggio di benvenuto per verificare che l'installazione abbia avuto successo.

Installazione di Docker su Windows

Requisiti: versione a 64 bit di Windows 7 o versioni successive su una macchina che supporta la tecnologia di virtualizzazione dell'hardware ed è abilitata.

Mentre il binario docker può essere eseguito in modo nativo su Windows, per costruire e ospitare i contenitori è necessario eseguire una macchina virtuale Linux sulla scatola.

1.12.0

Dalla versione 1.12 non è necessario installare una VM separata, poiché Docker può utilizzare la funzionalità nativa di Hyper-V di Windows per avviare una piccola macchina Linux che funga da backend.

Per installare la finestra mobile seguire i seguenti passi:

1. Vai a Docker per Windows
2. Scarica ed esegui il programma di installazione.
3. Continua attraverso il programma di installazione con le opzioni predefinite e inserisci le credenziali del tuo account quando richiesto.

Controlla qui per ulteriori informazioni sull'installazione.

1.11.2

Fino alla versione 1.11 il modo migliore per eseguire questa VM Linux è installare Docker Toolbox, che installa Docker, VirtualBox e la macchina guest Linux.

Per installare la casella degli strumenti finestra mobile, attenersi alla seguente procedura:

1. Vai a Docker Toolbox
2. Fare clic sul collegamento per Windows ed eseguire il programma di installazione.
3. Continua attraverso il programma di installazione con le opzioni predefinite e inserisci le credenziali del tuo account quando richiesto.

Questo installerà i binari Docker in Program Files e aggiornerà qualsiasi installazione di Virtual Box esistente. Controlla qui per ulteriori informazioni sull'installazione.

Per verificare l'installazione:

1.12.0

1. Avvia `Docker` dal menu Start se non è ancora stato avviato e assicurati che sia in esecuzione. Avanti up up su qualsiasi terminale (sia `cmd` o PowerShell)

1.11.2

1. Sul desktop, trova l'icona Docker Toolbox. Fare clic sull'icona per avviare un terminale Docker Toolbox.

2. Una volta che il terminale è di tipo aperto

```
docker run hello-world
```

3. Se tutto va bene, questo dovrebbe stampare un messaggio di benvenuto per verificare che l'installazione abbia avuto successo.

Installazione della finestra mobile su Ubuntu Linux

Docker è supportato nelle seguenti versioni a *64 bit* di Ubuntu Linux:

- Ubuntu Xenial 16.04 (LTS)
- Ubuntu Wily 15.10
- Ubuntu Trusty 14.04 (LTS)
- Ubuntu Precise 12.04 (LTS)

Un paio di note:

> Le seguenti istruzioni riguardano l'installazione usando solo i pacchetti **Docker** , e questo garantisce l'ottenimento dell'ultima versione ufficiale di **Docker** . Se è necessario installare solo utilizzando pacchetti `Ubuntu-managed` , consultare la documentazione di Ubuntu (non consigliato diversamente per ovvi motivi).
>
> Ubuntu Utopic 14.10 e 15.04 esistono nel repository APT di Docker ma non sono più supportati ufficialmente a causa di noti problemi di sicurezza.

Prerequisiti

- Docker funziona solo su un'installazione di Linux a 64 bit.
- Docker richiede il kernel Linux versione 3.10 o successiva (ad eccezione di `Ubuntu Precise 12.04` , che richiede la versione 3.13 o successiva). I kernel più vecchi di 3.10 non dispongono delle funzionalità necessarie per eseguire i contenitori Docker e contengono bug noti che causano la perdita di dati e spesso il panico in determinate condizioni. Controlla la versione attuale del kernel con il comando `uname -r` . Controlla questo post se hai bisogno di aggiornare il tuo kernel di `Ubuntu Precise (12.04 LTS)` scorrendo più in basso. Fai riferimento a questo post WikiHow per ottenere la versione più recente per altre installazioni di Ubuntu.

Aggiorna fonti APT

Questo deve essere fatto in modo tale da accedere ai pacchetti dal repository Docker.

1. Accedi al tuo computer come utente con privilegi `sudo` o `root` .
2. Apri una finestra di terminale.
3. Aggiorna le informazioni sul pacchetto, assicurati che APT funzioni con il metodo https e che i certificati CA siano installati.

```
$ sudo apt-get update
$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
```

4. Aggiungi la chiave GPG ufficiale di Docker:

   ```
   $ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
   ```

   Verificare che l'impronta digitale della chiave sia **9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88** .

   ```
   $ sudo apt-key fingerprint 0EBFCD88
   ```

```
      pub   4096R/0EBFCD88 2017-02-22
            Key fingerprint = 9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
      uid                  Docker Release (CE deb) <docker@docker.com>
      sub   4096R/F273FCD8 2017-02-22
```

5. Trova la voce nella tabella sottostante che corrisponde alla tua versione di Ubuntu. Questo determina dove APT cercherà i pacchetti Docker. Se possibile, esegui un'edizione di supporto a lungo termine (LTS) di Ubuntu.

Versione di Ubuntu	deposito
Preciso 12.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-precise main`
Trusty 14.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-trusty main`
Wily 15.10	`deb https://apt.dockerproject.org/repo ubuntu-wily main`
Xenial 16.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-xenial main`

   Nota: Docker non fornisce pacchetti per tutte le architetture. Gli artefatti binari vengono creati ogni notte e puoi scaricarli da `https://master.dockerproject.org` . Per installare la finestra mobile su un sistema multi-architettura, aggiungere una clausola `[arch=...]` alla voce. Fare riferimento al wiki di Debian Multiarch per i dettagli.

6. Eseguire il comando seguente, sostituendo la voce relativa al proprio sistema operativo per il segnaposto `<REPO>` .

   $ echo "" | sudo tee /etc/apt/sources.list.d/docker.list

7. Aggiorna l'indice del pacchetto `APT` eseguendo `sudo apt-get update` .

8. Verifica che `APT` stia prelevando dal repository corretto.

Quando si esegue il comando seguente, viene restituita una voce per ogni versione di Docker

disponibile per l'installazione. Ogni voce dovrebbe avere l'URL `https://apt.dockerproject.org/repo/` . La versione attualmente installata è contrassegnata con `***` . Vedi l'output dell'esempio qui sotto.

```
$ apt-cache policy docker-engine

  docker-engine:
    Installed: 1.12.2-0~trusty
    Candidate: 1.12.2-0~trusty
    Version table:
   *** 1.12.2-0~trusty 0
          500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
          100 /var/lib/dpkg/status
       1.12.1-0~trusty 0
          500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
       1.12.0-0~trusty 0
          500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
```

Da ora in poi quando esegui l' `apt-get upgrade` , `APT` preleva dal nuovo repository.

Prerequisiti di Ubuntu Version

Per Ubuntu Trusty (14.04), Wily (15.10) e Xenial (16.04), installare i pacchetti del kernel `linux-image-extra-*` , che consente di utilizzare il driver di archiviazione `aufs` .

Per installare i pacchetti `linux-image-extra-*` :

1. Apri un terminale sul tuo host Ubuntu.
2. Aggiorna il tuo gestore di pacchetti con il comando `sudo apt-get update` .
3. Installa i pacchetti consigliati.

   ```
   $ sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual
   ```

4. Procedere con l'installazione di Docker

Per Ubuntu Precise (12.04 LTS), Docker richiede la versione del kernel 3.13. Se la versione del tuo kernel è precedente alla 3.13, devi aggiornarla. Fare riferimento a questa tabella per vedere quali pacchetti sono necessari per il proprio ambiente:

Pacchetto	Descrizione
`linux-image-generic-lts-trusty`	Immagine del kernel Linux generico. Questo kernel ha `AUFS` integrato. È necessario per eseguire Docker.
`linux-headers-generic-lts-trusty`	Consente pacchetti come `VirtualBox guest additions ZFS` e `VirtualBox guest additions` che dipendono da loro. Se non hai installato le intestazioni per il tuo kernel esistente, puoi saltare queste intestazioni per il kernel `trusty` . Se non sei sicuro, dovresti includere questo pacchetto per sicurezza.
`xserver-xorg-`	Opzionale in ambienti non grafici senza Unity / Xorg. **Obbligatorio** quando

Pacchetto	Descrizione
`lts-trusty`	si esegue Docker sulla macchina con un ambiente grafico.
`libgl1-mesa-glx-lts-trusty`	Per saperne di più sui motivi di questi pacchetti, leggi le istruzioni di installazione per i kernel backport, in particolare lo Stack di abilitazione LTS . Fare riferimento alla nota 5 sotto ogni versione.

Per aggiornare il kernel e installare i pacchetti aggiuntivi, effettuare le seguenti operazioni:

1. Apri un terminale sul tuo host Ubuntu.
2. Aggiorna il tuo gestore di pacchetti con il comando `sudo apt-get update` .
3. Installa entrambi i pacchetti richiesti e opzionali.

```
$ sudo apt-get install linux-image-generic-lts-trusty
```

4. Ripeti questo passaggio per altri pacchetti che devi installare.
5. Riavvia il tuo host per usare il kernel aggiornato usando il comando `sudo reboot` .
6. Dopo il riavvio, vai avanti e installa Docker.

Installa l'ultima versione

Assicurati di soddisfare i prerequisiti, solo allora segui i passaggi seguenti.

> **Nota:** per i sistemi di produzione, si consiglia di installare una versione specifica in modo da non aggiornare accidentalmente Docker. È necessario pianificare attentamente gli aggiornamenti per i sistemi di produzione.

1. Accedi alla tua installazione di Ubuntu come utente con privilegi `sudo` . (Possibilmente eseguendo `sudo -su`).
2. Aggiorna l'indice del tuo pacchetto APT eseguendo `sudo apt-get update` .
3. Installa Docker Community Edition con il comando `sudo apt-get install docker-ce` .
4. Avviare il daemon `docker` con il comando `sudo service docker start` .
5. Verificare che la `docker` sia installata correttamente eseguendo l'immagine hello-world.

```
$ sudo docker run hello-world
```

Questo comando scarica un'immagine di prova e la esegue in un contenitore. Quando il contenitore viene eseguito, stampa un messaggio informativo ed esce.

Gestisci Docker come utente non root

Se non si desidera utilizzare `sudo` quando si utilizza il comando finestra mobile, creare un gruppo

Unix chiamato `docker` e aggiungere utenti ad esso. Quando il daemon `docker` viene `docker` , rende la proprietà del socket Unix leggibile / scrivibile dal gruppo docker.

Per creare il gruppo `docker` e aggiungere l'utente:

1. Accedi ad Ubuntu come utente con privilegi `sudo` .
2. Creare il gruppo `docker` con il comando `sudo groupadd docker` .
3. Aggiungi il tuo utente al gruppo `docker` .

```
$ sudo usermod -aG docker $USER
```

4. Disconnettersi e riconnettersi in modo che l'appartenenza al gruppo venga rivalutata.
5. Verificare di poter eseguire i comandi di `docker` senza autorizzazione `sudo` .

```
$ docker run hello-world
```

Se fallisce, vedrai un errore:

```
Cannot connect to the Docker daemon. Is 'docker daemon' running on this host?
```

Controlla se la variabile d'ambiente `DOCKER_HOST` è impostata per la tua shell.

```
$ env | grep DOCKER_HOST
```

Se è impostato, il comando precedente restituirà un risultato. Se è così, disattivalo.

```
$ unset DOCKER_HOST
```

Potrebbe essere necessario modificare il proprio ambiente in file come `~/.bashrc` o `~/.profile` per impedire che la variabile `DOCKER_HOST` venga impostata erroneamente.

Installare Docker su Ubuntu

Requisiti: Docker può essere installato su qualsiasi Linux con un kernel di almeno versione 3.10. Docker è supportato nelle seguenti versioni a 64 bit di Ubuntu Linux:

- Ubuntu Xenial 16.04 (LTS)
- Ubuntu Wily 15.10
- Ubuntu Trusty 14.04 (LTS)
- Ubuntu Precise 12.04 (LTS)

Installazione facile

Nota: l'installazione di Docker dal repository di Ubuntu predefinito installerà una versione precedente di Docker.

Per installare l'ultima versione di Docker utilizzando il repository Docker, usa `curl` per afferrare ed eseguire lo script di installazione fornito da Docker:

```
$ curl -sSL https://get.docker.com/ | sh
```

In alternativa, è possibile utilizzare `wget` per installare Docker:

```
$ wget -qO- https://get.docker.com/ | sh
```

Docker verrà ora installato.

Installazione manuale

Se, tuttavia, l'esecuzione dello script di installazione non è un'opzione, è possibile utilizzare le seguenti istruzioni per installare manualmente l'ultima versione di Docker dal repository ufficiale.

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https ca-certificates
```

Aggiungi la chiave GPG:

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 \
  --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

Quindi, apri il file `/etc/apt/sources.list.d/docker.list` nel tuo editor preferito. Se il file non esiste, crearlo. Rimuovi eventuali voci esistenti. Quindi, in base alla tua versione, aggiungi la seguente riga:

- Ubuntu Precise 12.04 (LTS):

  ```
  deb https://apt.dockerproject.org/repo ubuntu-precise main
  ```

- Ubuntu Trusty 14.04 (LTS)

  ```
  deb https://apt.dockerproject.org/repo ubuntu-trusty main
  ```

- Ubuntu Wily 15.10

  ```
  deb https://apt.dockerproject.org/repo ubuntu-wily main
  ```

- Ubuntu Xenial 16.04 (LTS)

  ```
  deb https://apt.dockerproject.org/repo ubuntu-xenial main
  ```

Salva il file ed esci, quindi aggiorna l'indice del pacchetto, disinstalla tutte le versioni installate di Docker e verifica che `apt` stia tirando dal repository corretto:

```
$ sudo apt-get update
$ sudo apt-get purge lxc-docker
$ sudo apt-cache policy docker-engine
```

A seconda della versione di Ubuntu, potrebbero essere necessari alcuni prerequisiti:

- Ubuntu Xenial 16.04 (LTS), Ubuntu Wily 15.10, Ubuntu Trusty 14.04 (LTS)

  ```
  sudo apt-get update && sudo apt-get install linux-image-extra-$(uname -r)
  ```

- Ubuntu Precise 12.04 (LTS)

  Questa versione di Ubuntu richiede la versione 3.13 del kernel. Potrebbe essere necessario installare pacchetti aggiuntivi a seconda del proprio ambiente:

  ```
  linux-image-generic-lts-trusty
  ```

  Immagine del kernel Linux generico. Questo kernel ha AUFS integrato. È necessario per eseguire Docker.

  ```
  linux-headers-generic-lts-trusty
  ```

  Consente pacchetti come le aggiunte guest ZFS e VirtualBox che dipendono da loro. Se non hai installato le intestazioni per il tuo kernel esistente, puoi saltare queste intestazioni per il kernel `trusty` . Se non sei sicuro, dovresti includere questo pacchetto per sicurezza.

  ```
  xserver-xorg-lts-trusty
  ```

  ```
  libgl1-mesa-glx-lts-trusty
  ```

  Questi due pacchetti sono opzionali in ambienti non grafici senza Unity / Xorg. Obbligatorio quando si esegue Docker sulla macchina con un ambiente grafico.

  Per saperne di più sui motivi di questi pacchetti, leggi le istruzioni di installazione per i kernel backport, in particolare lo Stack di abilitazione LTS - fai riferimento alla nota 5 sotto ogni versione.

  Installa i pacchetti richiesti, quindi riavvia l'host:

  ```
  $ sudo apt-get install linux-image-generic-lts-trusty
  ```

  ```
  $ sudo reboot
  ```

Infine, aggiorna l'indice del pacchetto `apt` e installa Docker:

```
$ sudo apt-get update
$ sudo apt-get install docker-engine
```

Avvia il demone:

```
$ sudo service docker start
```

Ora verifica che la finestra mobile funzioni correttamente avviando un'immagine di prova:

```
$ sudo docker run hello-world
```

Questo comando dovrebbe stampare un messaggio di benvenuto per verificare che l'installazione

abbia avuto successo.

Crea un contenitore finestra mobile in Google Cloud

È possibile utilizzare la finestra mobile, senza utilizzare il daemon docker (motore), utilizzando i provider cloud. In questo esempio, dovresti avere un `gcloud` (Google Cloud util), collegato al tuo account

```
docker-machine create --driver google --google-project `your-project-name` google-machine-type f1-large fm02
```

Questo esempio creerà una nuova istanza nella tua Google Cloud Console. Utilizzo del tempo macchina `f1-large`

Installa Docker su Ubuntu

Docker è supportato nelle seguenti versioni a *64 bit* di Ubuntu Linux:

- Ubuntu Xenial 16.04 (LTS)
- Ubuntu Wily 15.10
- Ubuntu Trusty 14.04 (LTS)
- Ubuntu Precise 12.04 (LTS)

Un paio di note:

> Le seguenti istruzioni riguardano l'installazione usando solo i pacchetti **Docker** , e questo garantisce l'ottenimento dell'ultima versione ufficiale di **Docker** . Se è necessario installare solo utilizzando pacchetti `Ubuntu-managed` , consultare la documentazione di Ubuntu (non consigliato diversamente per ovvi motivi).
>
> Ubuntu Utopic 14.10 e 15.04 esistono nel repository APT di Docker ma non sono più supportati ufficialmente a causa di noti problemi di sicurezza.

Prerequisiti

- Docker funziona solo su un'installazione di Linux a 64 bit.
- Docker richiede il kernel Linux versione 3.10 o successiva (ad eccezione di `Ubuntu Precise 12.04` , che richiede la versione 3.13 o successiva). I kernel più vecchi di 3.10 non dispongono delle funzionalità necessarie per eseguire i contenitori Docker e contengono bug noti che causano la perdita di dati e spesso il panico in determinate condizioni. Controlla la versione attuale del kernel con il comando `uname -r` . Controlla questo post se hai bisogno di aggiornare il tuo kernel di `Ubuntu Precise (12.04 LTS)` scorrendo più in basso. Fai riferimento a questo post WikiHow per ottenere la versione più recente per altre installazioni di Ubuntu.

Aggiorna fonti APT

Questo deve essere fatto in modo tale da accedere ai pacchetti dal repository Docker.

1. Accedi al tuo computer come utente con privilegi `sudo` o `root` .

2. Apri una finestra di terminale.
3. Aggiorna le informazioni sul pacchetto, assicurati che APT funzioni con il metodo https e che i certificati CA siano installati.

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https ca-certificates
```

4. Aggiungi la nuova chiave `GPG` . Questo comando scarica la chiave con l'ID `58118E89F3A912897C070ADBF76221572C52609D` dal server delle `hkp://ha.pool.sks-keyservers.net:80` e la aggiunge al `adv keychain` . Per ulteriori informazioni, vedere l'output di `man apt-key` .

   ```
   $ sudo apt-key adv \
          --keyserver hkp://ha.pool.sks-keyservers.net:80 \
          --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
   ```

5. Trova la voce nella tabella sottostante che corrisponde alla tua versione di Ubuntu. Questo determina dove APT cercherà i pacchetti Docker. Se possibile, esegui un'edizione di supporto a lungo termine (LTS) di Ubuntu.

Versione di Ubuntu	deposito
Preciso 12.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-precise main`
Trusty 14.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-trusty main`
Wily 15.10	`deb https://apt.dockerproject.org/repo ubuntu-wily main`
Xenial 16.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-xenial main`

   Nota: Docker non fornisce pacchetti per tutte le architetture. Gli artefatti binari vengono creati ogni notte e puoi scaricarli da `https://master.dockerproject.org` . Per installare la finestra mobile su un sistema multi-architettura, aggiungere una clausola `[arch=...]` alla voce. Fare riferimento al wiki di Debian Multiarch per i dettagli.

6. Eseguire il comando seguente, sostituendo la voce relativa al proprio sistema operativo per il segnaposto `<REPO>` .

   $ echo "" | sudo tee /etc/apt/sources.list.d/docker.list

7. Aggiorna l'indice del pacchetto `APT` eseguendo `sudo apt-get update` .

8. Verifica che `APT` stia prelevando dal repository corretto.

Quando si esegue il comando seguente, viene restituita una voce per ogni versione di Docker disponibile per l'installazione. Ogni voce dovrebbe avere l'URL `https://apt.dockerproject.org/repo/` . La versione attualmente installata è contrassegnata con `***` . Vedi l'output dell'esempio qui sotto.

```
$ apt-cache policy docker-engine

  docker-engine:
    Installed: 1.12.2-0~trusty
    Candidate: 1.12.2-0~trusty
    Version table:
   *** 1.12.2-0~trusty 0
          500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
          100 /var/lib/dpkg/status
       1.12.1-0~trusty 0
          500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
       1.12.0-0~trusty 0
          500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
```

Da ora in poi quando esegui l' `apt-get upgrade` , `APT` preleva dal nuovo repository.

Prerequisiti di Ubuntu Version

Per Ubuntu Trusty (14.04), Wily (15.10) e Xenial (16.04), installare i pacchetti del kernel `linux-image-extra-*` , che consente di utilizzare il driver di archiviazione `aufs` .

Per installare i pacchetti `linux-image-extra-*` :

1. Apri un terminale sul tuo host Ubuntu.
2. Aggiorna il tuo gestore di pacchetti con il comando `sudo apt-get update` .
3. Installa i pacchetti consigliati.

   ```
   $ sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual
   ```

4. Procedere con l'installazione di Docker

Per Ubuntu Precise (12.04 LTS), Docker richiede la versione del kernel 3.13. Se la versione del tuo kernel è precedente alla 3.13, devi aggiornarla. Fare riferimento a questa tabella per vedere quali pacchetti sono necessari per il proprio ambiente:

Pacchetto	Descrizione
`linux-image-generic-lts-trusty`	Immagine del kernel Linux generico. Questo kernel ha `AUFS` integrato. È necessario per eseguire Docker.
`linux-headers-generic-lts-trusty`	Consente pacchetti come `VirtualBox guest additions ZFS` e `VirtualBox guest additions` che dipendono da loro. Se non hai installato le intestazioni per il tuo kernel esistente, puoi saltare queste intestazioni per il kernel `trusty` . Se non sei sicuro, dovresti includere questo pacchetto per sicurezza.
`xserver-xorg-lts-trusty`	Opzionale in ambienti non grafici senza Unity / Xorg. **Obbligatorio** quando si esegue Docker sulla macchina con un ambiente grafico.
`ligbl1-mesa-glx-lts-trusty`	Per saperne di più sui motivi di questi pacchetti, leggi le istruzioni di installazione per i kernel backport, in particolare lo Stack di abilitazione LTS

Pacchetto	Descrizione
	. Fare riferimento alla nota 5 sotto ogni versione.

Per aggiornare il kernel e installare i pacchetti aggiuntivi, effettuare le seguenti operazioni:

1. Apri un terminale sul tuo host Ubuntu.
2. Aggiorna il tuo gestore di pacchetti con il comando `sudo apt-get update` .
3. Installa entrambi i pacchetti richiesti e opzionali.

```
$ sudo apt-get install linux-image-generic-lts-trusty
```

4. Ripeti questo passaggio per altri pacchetti che devi installare.
5. Riavvia il tuo host per usare il kernel aggiornato usando il comando `sudo reboot` .
6. Dopo il riavvio, vai avanti e installa Docker.

Installa l'ultima versione

Assicurati di soddisfare i prerequisiti, solo allora segui i passaggi seguenti.

> **Nota:** per i sistemi di produzione, si consiglia di installare una versione specifica in modo da non aggiornare accidentalmente Docker. È necessario pianificare attentamente gli aggiornamenti per i sistemi di produzione.

1. Accedi alla tua installazione di Ubuntu come utente con privilegi `sudo` . (Possibilmente eseguendo `sudo -su`).
2. Aggiorna l'indice del tuo pacchetto APT eseguendo `sudo apt-get update` .
3. Installa Docker con il comando `sudo apt-get install docker-engine` .
4. Avviare il daemon `docker` con il comando `sudo service docker start` .
5. Verificare che la `docker` sia installata correttamente eseguendo l'immagine hello-world.

```
$ sudo docker run hello-world
```

Questo comando scarica un'immagine di prova e la esegue in un contenitore. Quando il contenitore viene eseguito, stampa un messaggio informativo ed esce.

Gestisci Docker come utente non root

Se non si desidera utilizzare `sudo` quando si utilizza il comando finestra mobile, creare un gruppo Unix chiamato `docker` e aggiungere utenti ad esso. Quando il daemon `docker` viene `docker` , rende la proprietà del socket Unix leggibile / scrivibile dal gruppo docker.

Per creare il gruppo `docker` e aggiungere l'utente:

1. Accedi ad Ubuntu come utente con privilegi `sudo` .
2. Creare il gruppo `docker` con il comando `sudo groupadd docker` .
3. Aggiungi il tuo utente al gruppo `docker` .

```
$ sudo usermod -aG docker $USER
```

4. Disconnettersi e riconnettersi in modo che l'appartenenza al gruppo venga rivalutata.
5. Verificare di poter eseguire i comandi di `docker` senza autorizzazione `sudo` .

```
$ docker run hello-world
```

Se fallisce, vedrai un errore:

```
Cannot connect to the Docker daemon. Is 'docker daemon' running on this host?
```

Controlla se la variabile d'ambiente `DOCKER_HOST` è impostata per la tua shell.

```
$ env | grep DOCKER_HOST
```

Se è impostato, il comando precedente restituirà un risultato. Se è così, disattivalo.

```
$ unset DOCKER_HOST
```

Potrebbe essere necessario modificare il proprio ambiente in file come `~/.bashrc` o `~/.profile` per impedire che la variabile `DOCKER_HOST` venga impostata erroneamente.

Installazione di Docker-ce OR Docker-ee su CentOS

Docker ha annunciato le seguenti edizioni:

-Docker-ee (Enterprise Edition) insieme a Docker-ce (Community Edition) e Docker (Supporto commerciale)

Questo documento ti aiuterà con le fasi di installazione di Docker-ee e Docker-ce edition in CentOS

Installazione Docker-ce

Di seguito sono riportati i passaggi per installare l'edizione docker-ce

1. Installa yum-utils, che fornisce l'utility yum-config-manager:

```
$ sudo yum install -y yum-utils
```

2. Utilizzare il seguente comando per configurare il repository stabile:

```
$ sudo yum-config-manager \
 --add-repo \
 https://download.docker.com/linux/centos/docker-ce.repo
```

3. Opzionale: abilita il repository edge. Questo repository è incluso nel file docker.repo precedente ma è disabilitato per impostazione predefinita. Puoi abilitarlo accanto al repository stabile.

```
 $ sudo yum-config-manager --enable docker-ce-edge
```

- È possibile disabilitare il repository edge eseguendo il comando `yum-config-manager` con il flag `--disable` . Per riattivarlo, usa il flag `--enable` . Il seguente comando disabilita il repository edge.

```
 $ sudo yum-config-manager --disable docker-ce-edge
```

4. Aggiorna l'indice del pacchetto yum.

```
 $ sudo yum makecache fast
```

5. Installa il docker-ce usando il seguente comando:

```
 $ sudo yum install docker-ce-17.03.0.ce
```

6. Conferma l'impronta digitale di Docker-ce

   `060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35`

   Se si desidera installare un'altra versione di docker-ce, è possibile utilizzare il seguente comando:

```
$ sudo yum install docker-ce-VERSION
```

   Specificare il numero di `VERSION`

7. Se tutto è andato bene, docker-ce è ora installato nel tuo sistema, usa il seguente comando per iniziare:

```
 $ sudo systemctl start docker
```

8. Verifica l'installazione della finestra mobile:

```
 $ sudo docker run hello-world
```

   dovresti ottenere il seguente messaggio:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

-Docker-ee (Enterprise Edition) Installazione

Per Enterprise Edition (EE) sarebbe necessario registrarsi, per ottenere il <DOCKER-EE-URL>.

1. Per iscriverti vai su https://cloud.docker.com/ . Inserisci i tuoi dati e conferma il tuo ID e-mail. Dopo la conferma ti verrà dato un <DOCKER-EE-URL>, che puoi vedere nella tua dashboard dopo aver fatto clic su setup.

2. Rimuovi eventuali repository Docker esistenti da `/etc/yum.repos.d/`

3. Memorizza l'URL del tuo repository EE Docker in una variabile yum in `/etc/yum/vars/` . Sostituisci <DOCKER-EE-URL> con l'URL annotato nel primo passaggio.

   ```
   $ sudo sh -c 'echo "<DOCKER-EE-URL>" > /etc/yum/vars/dockerurl'
   ```

4. Installa yum-utils, che fornisce l'utilità yum-config-manager:

   ```
   $ sudo yum install -y yum-utils
   ```

5. Utilizzare il seguente comando per aggiungere il repository stabile:

   ```
   $ sudo yum-config-manager \
   --add-repo \
   <DOCKER-EE-URL>/docker-ee.repo
   ```

6. Aggiorna l'indice del pacchetto yum.

   ```
   $ sudo yum makecache fast
   ```

7. Installa docker-ee

   ```
   sudo yum install docker-ee
   ```

8. È possibile avviare la finestra mobile-ee utilizzando il seguente comando:

   ```
   $ sudo systemctl start docker
   ```

Leggi Iniziare con Docker online: https://riptutorial.com/it/docker/topic/658/iniziare-con-docker

Capitolo 2: API del motore Docker

introduzione

Un'API che ti consente di controllare ogni aspetto di Docker dalle tue applicazioni, creare strumenti per gestire e monitorare le applicazioni in esecuzione su Docker e persino usarlo per creare app su Docker stesso.

Examples

Abilita l'accesso remoto all'API Docker su Linux

Modifica `/etc/init/docker.conf` e aggiorna la variabile `DOCKER_OPTS` al seguente:

```
DOCKER_OPTS='-H tcp://0.0.0.0:4243 -H unix:///var/run/docker.sock'
```

Riavvia Deamon Docker

```
service docker restart
```

Verifica se l'API remota funziona

```
curl -X GET http://localhost:4243/images/json
```

Abilita l'accesso remoto all'API Docker su Linux con sistema systemd

Linux con systemd, come Ubuntu 16.04, aggiungendo `-H tcp://0.0.0.0:2375` a `/etc/default/docker` non ha l'effetto a cui è abituato.

Invece, crea un file chiamato `/etc/systemd/system/docker-tcp.socket` per rendere disponibile la finestra mobile su un socket TCP sulla porta 4243:

```
[Unit]
Description=Docker Socket for the API
[Socket]
ListenStream=4243
Service=docker.service
[Install]
WantedBy=sockets.target
```

Quindi abilita il nuovo socket:

```
systemctl enable docker-tcp.socket
systemctl enable docker.socket
systemctl stop docker
systemctl start docker-tcp.socket
systemctl start docker
```

Ora verifica se l'API remota funziona:

```
curl -X GET http://localhost:4243/images/json
```

Abilita l'accesso remoto con TLS su Systemd

Copiare il file dell'unità di installazione del pacchetto su / etc dove le modifiche non verranno sovrascritte su un aggiornamento:

```
cp /lib/systemd/system/docker.service /etc/systemd/system/docker.service
```

Aggiorna /etc/systemd/system/docker.service con le tue opzioni su ExecStart:

```
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:2376 \
  --tlsverify --tlscacert=/etc/docker/certs/ca.pem \
  --tlskey=/etc/docker/certs/key.pem \
  --tlscert=/etc/docker/certs/cert.pem
```

Si noti che `dockerd` è il nome del daemon 1.12, prima era `docker daemon` . Si noti inoltre che 2376 è la porta TLS standard per i docker, 2375 è la porta standard non crittografata. Vedere questa pagina per i passaggi per creare la propria CA autofirmata TLS, certificato e chiave.

Dopo aver apportato le modifiche ai file dell'unità systemd, eseguire quanto segue per ricaricare systemd config:

```
systemctl daemon-reload
```

Quindi eseguire quanto segue per riavviare la finestra mobile:

```
systemctl restart docker
```

È una cattiva idea ignorare la crittografia TLS quando si espone la porta Docker poiché chiunque abbia accesso alla rete a questa porta ha effettivamente accesso completo alla radice sull'host.

Immagine che tira con le barre di avanzamento, scritte in Go

Ecco un esempio di immagine che tira usando le `Docker Engine API Go` e `Docker Engine API` e le stesse barre di avanzamento di quelle mostrate quando si esegue la `docker pull your_image_name` nella `CLI` . Ai fini delle barre di avanzamento vengono utilizzati alcuni codici ANSI .

```
package yourpackage

import (
    "context"
    "encoding/json"
    "fmt"
    "io"
    "strings"
```

```
    "github.com/docker/docker/api/types"
    "github.com/docker/docker/client"
)

// Struct representing events returned from image pulling
type pullEvent struct {
    ID             string `json:"id"`
    Status         string `json:"status"`
    Error          string `json:"error,omitempty"`
    Progress       string `json:"progress,omitempty"`
    ProgressDetail struct {
        Current int `json:"current"`
        Total   int `json:"total"`
    } `json:"progressDetail"`
}

// Actual image pulling function
func PullImage(dockerImageName string) bool {
    client, err := client.NewEnvClient()

    if err != nil {
        panic(err)
    }

    resp, err := client.ImagePull(context.Background(), dockerImageName,
types.ImagePullOptions{})

    if err != nil {
        panic(err)
    }

    cursor := Cursor{}
    layers := make([]string, 0)
    oldIndex := len(layers)

    var event *pullEvent
    decoder := json.NewDecoder(resp)

    fmt.Printf("\n")
    cursor.hide()

    for {
        if err := decoder.Decode(&event); err != nil {
            if err == io.EOF {
                break
            }

            panic(err)
        }

        imageID := event.ID

        // Check if the line is one of the final two ones
        if strings.HasPrefix(event.Status, "Digest:") || strings.HasPrefix(event.Status,
"Status:") {
            fmt.Printf("%s\n", event.Status)
            continue
        }

        // Check if ID has already passed once
        index := 0
```

```
        for i, v := range layers {
            if v == imageID {
                index = i + 1
                break
            }
        }

        // Move the cursor
        if index > 0 {
            diff := index - oldIndex

            if diff > 1 {
                down := diff - 1
                cursor.moveDown(down)
            } else if diff < 1 {
                up := diff*(-1) + 1
                cursor.moveUp(up)
            }

            oldIndex = index
        } else {
            layers = append(layers, event.ID)
            diff := len(layers) - oldIndex

            if diff > 1 {
                cursor.moveDown(diff) // Return to the last row
            }

            oldIndex = len(layers)
        }

        cursor.clearLine()

        if event.Status == "Pull complete" {
            fmt.Printf("%s: %s\n", event.ID, event.Status)
        } else {
            fmt.Printf("%s: %s %s\n", event.ID, event.Status, event.Progress)
        }

    }

    cursor.show()

    if strings.Contains(event.Status, fmt.Sprintf("Downloaded newer image for %s",
dockerImageName)) {
        return true
    }

    return false
}
```

Per una migliore leggibilità, le azioni del cursore con i codici ANSI vengono spostate in una struttura separata, che assomiglia a questo:

```
package yourpackage

import "fmt"

// Cursor structure that implements some methods
// for manipulating command line's cursor
```

```
type Cursor struct{}

func (cursor *Cursor) hide() {
    fmt.Printf("\033[?25l")
}

func (cursor *Cursor) show() {
    fmt.Printf("\033[?25h")
}

func (cursor *Cursor) moveUp(rows int) {
    fmt.Printf("\033[%dF", rows)
}

func (cursor *Cursor) moveDown(rows int) {
    fmt.Printf("\033[%dE", rows)
}

func (cursor *Cursor) clearLine() {
    fmt.Printf("\033[2K")
}
```

Dopodiché nel tuo pacchetto principale puoi chiamare la funzione `PullImage` passando il nome dell'immagine che vuoi estrarre. Naturalmente, prima di chiamarlo, è necessario accedere al registro Docker, dove si trova l'immagine.

Fare una richiesta CURL con il passaggio di alcune strutture complesse

Quando si utilizza `cURL` per alcune query `Docker API` , potrebbe essere un po 'complicato passare alcune strutture complesse. Diciamo che ottenere un elenco di immagini consente di utilizzare i filtri come parametro di query, che deve essere una rappresentazione `JSON` della `map[string][]string` della mappa (sulle mappe in `Go` puoi trovare altre informazioni qui).
Ecco come ottenere questo:

```
curl --unix-socket /var/run/docker.sock \
    -XGET "http:/v1.29/images/json" \
    -G \
    --data-urlencode 'filters={"reference":{"yourpreciousregistry.com/path/to/image": true},
"dangling":{"true": true}}'
```

Qui viene utilizzato il flag `-G` per specificare che i dati nel parametro `--data-urlencode` verranno utilizzati in una richiesta `HTTP GET` invece della richiesta `POST` che altrimenti verrebbe utilizzata. I dati verranno aggiunti all'URL con un `?` separatore.

Leggi API del motore Docker online: https://riptutorial.com/it/docker/topic/3935/api-del-motore-docker

Capitolo 3: Checkpoint e ripristino dei contenitori

Examples

Compilazione finestra mobile con checkpoint e ripristino abilitato (ubuntu)

Per compilare la finestra mobile è consigliabile avere almeno **2 GB di RAM** . Anche se a volte fallisce, è meglio usare **4GB** .

1. assicurati che git e make siano installati

```
sudo apt-get install make git-core -y
```

2. installa un nuovo kernel (almeno 4.2)

```
sudo apt-get install linux-generic-lts-xenial
```

3. riavviare il computer per attivare il nuovo kernel

```
sudo reboot
```

4. compilare `criu` che è necessario per eseguire il `docker checkpoint`

```
sudo apt-get install libprotobuf-dev libprotobuf-c0-dev protobuf-c-compiler protobuf-
compiler python-protobuf libnl-3-dev libcap-dev -y
wget http://download.openvz.org/criu/criu-2.4.tar.bz2 -O - | tar -xj
cd criu-2.4
make
make install-lib
make install-criu
```

5. controlla se tutti i requisiti sono soddisfatti per eseguire criu

```
sudo criu check
```

6. compilare la finestra mobile sperimentale (abbiamo bisogno di finestra mobile per compilare la finestra mobile)

```
cd ~
wget -qO- https://get.docker.com/ | sh
sudo usermod -aG docker $(whoami)
```

- **A questo punto dobbiamo disconnetterci e riconnetterci per avere un demone docker. Dopo il relog continua con il passo di compilazione**

```
git clone https://github.com/boucher/docker
cd docker
git checkout docker-checkpoint-restore
make #that will take some time - drink a coffee
DOCKER_EXPERIMENTAL=1 make binary
```

7. Ora abbiamo una finestra mobile compilata. Consente di spostare i file binari. Assicurati di sostituire `<version>` con la versione installata

```
sudo service docker stop
sudo cp $(which docker) $(which docker)_ ; sudo cp ./bundles/latest/binary-client/docker-
<version>-dev $(which docker)
sudo cp $(which docker-containerd) $(which docker-containerd)_ ; sudo cp
./bundles/latest/binary-daemon/docker-containerd $(which docker-containerd)
sudo cp $(which docker-containerd-ctr) $(which docker-containerd-ctr)_ ; sudo cp
./bundles/latest/binary-daemon/docker-containerd-ctr $(which docker-containerd-ctr)
sudo cp $(which docker-containerd-shim) $(which docker-containerd-shim)_ ; sudo cp
./bundles/latest/binary-daemon/docker-containerd-shim $(which docker-containerd-shim)
sudo cp $(which dockerd) $(which dockerd)_ ; sudo cp ./bundles/latest/binary-
daemon/dockerd $(which dockerd)
sudo cp $(which docker-runc) $(which docker-runc)_ ; sudo cp ./bundles/latest/binary-
daemon/docker-runc $(which docker-runc)
sudo service docker start
```

Non preoccuparti: abbiamo eseguito il backup dei vecchi binari. Sono ancora lì ma con un trattino basso aggiunto al suo nome (`docker_`).

Congratulazioni ora hai una finestra mobile sperimentale con la possibilità di controllare un container e ripristinarlo.

Si noti che le funzioni sperimentali NON sono pronte per la produzione

Punto di controllo e ripristino di un contenitore

```
# create docker container
export cid=$(docker run -d --security-opt seccomp:unconfined busybox /bin/sh -c 'i=0; while
true; do echo $i; i=$(expr $i + 1); sleep 1; done')

# container is started and prints a number every second
# display the output with
docker logs $cid

# checkpoint the container
docker checkpoint create $cid checkpointname

# container is not running anymore
docker np

# lets pass some time to make sure

# resume container
docker start $cid --checkpoint=checkpointname

# print logs again
docker logs $cid
```

Leggi Checkpoint e ripristino dei contenitori online:
https://riptutorial.com/it/docker/topic/5291/checkpoint-e-ripristino-dei-contenitori

Capitolo 4: Collegamento di contenitori

Parametri

Parametro	Dettagli
`tty:true`	In docker-compose.yml, la bandiera `tty: true` mantiene attivo il comando `sh` del contenitore in attesa di input.

Osservazioni

I driver di rete `host` e `bridge` sono in grado di connettere i contenitori su un singolo host di docker. Per consentire ai contenitori di comunicare oltre una macchina, creare una rete di sovrapposizione. I passaggi per creare la rete dipendono da come vengono gestiti gli host della docker.

- Modalità `docker network create --driver overlay` : la `docker network create --driver overlay`
- finestra mobile / sciame : richiede un archivio di valori-chiave esterno

Examples

Docker network

I contenitori nella stessa rete mobile hanno accesso alle porte esposte.

```
docker network create sample
docker run --net sample --name keys consul agent -server -client=0.0.0.0 -bootstrap
```

Il Dockerfile di Console espone `8500` , `8600` e molte altre porte. Per dimostrare, esegui un altro contenitore nella stessa rete:

```
docker run --net sample -ti alpine sh
/ # wget -qO- keys:8500/v1/catalog/nodes
```

Qui il contenitore del console è risolto da `keys` , il nome dato nel primo comando. Docker fornisce la risoluzione DNS su questa rete, per trovare i contenitori con il loro `--name` .

Docker-composizione

Le reti possono essere specificate in un file di composizione (v2). Per impostazione predefinita tutti i contenitori si trovano in una rete condivisa.

Inizia con questo file: `example/docker-compose.yml` :

```
version: '2'
services:
  keys:
    image: consul
    command: agent -server -client=0.0.0.0 -bootstrap
  test:
    image: alpine
    tty: true
    command: sh
```

L'avvio di questo stack con `docker-compose up -d` creerà una rete che prende il nome dalla directory padre, in questo caso `example_default` . Controllare con la `docker network ls`

```
 > docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
719eafa8690b        example_default     bridge              local
```

Connettiti al contenitore alpino per verificare che i contenitori possano risolvere e comunicare:

```
 > docker exec -ti example_test_1 sh
/ # nslookup keys
...
/ # wget -qO- keys:8500/v1/kv/?recurse
...
```

Un file di composizione può avere una `networks:` sezione di livello superiore per specificare il nome della rete, il driver e altre opzioni dal comando di rete della finestra mobile .

Collegamento di container

L'argomento `link: --link` e il `link:` sezioni docker-compose creano *alias* in altri contenitori.

```
docker network create sample
docker run -d --net sample --name redis redis
```

Con il collegamento, il nome originale o la mappatura risolverà il contenitore redis.

```
> docker run --net sample --link redis:cache -ti python:alpine sh -c "pip install redis &&
python"
>>> import redis
>>> r = redis.StrictRedis(host='cache')
>>> r.set('key', 'value')
True
```

Prima di collegare il `1.10.0` contenitore `1.10.0` anche la connettività di rete - comportamento ora fornito dalla rete mobile. I collegamenti nelle versioni successive forniscono solo un effetto `legacy` sulla rete bridge predefinita.

Leggi Collegamento di contenitori online: https://riptutorial.com/it/docker/topic/6528/collegamento-di-contenitori

Capitolo 5: Come configurare la replica Mongo a tre nodi utilizzando l'immagine di Docker e Provisioned utilizzando Chef

introduzione

Questa documentazione descrive come creare un set di repliche Mongo a tre nodi utilizzando Docker Image e provisioning automatico utilizzando Chef.

Examples

Costruisci il passaggio

passi:

1. Genera un file di chiavi Base 64 per l'autenticazione del nodo Mongo. Metti questo file in chef data_bags
2. Vai al suppermarket dello chef e scarica il libro di ricette. Genera un libro di cucina personalizzato (ad es. Custom_mongo) e aggiungi "docker" dipendente, "~> 2.0" al metadata.rb del tuo libro di cucina
3. Crea attributi e ricette nel tuo ricettario personalizzato
4. Inizializzare Mongo per formare il cluster Set Set

Passaggio 1: crea il file chiave

crea data_bag chiamato mongo-keyfile e item chiamato keyfile. Questo sarà nella directory data_bags nello chef. Il contenuto dell'articolo sarà il seguente

```
openssl rand -base64 756 > <path-to-keyfile>
```

contenuto dell'articolo keyfile

```
{
  "id": "keyfile",
  "comment": "Mongo Repset keyfile",
  "key-file": "generated base 64 key above"
}
```

Passo 2: Scarica il ricettario di docker dal mercato dello chef e poi crea il libro di cucina custom_mongo

```
knife cookbook site download docker
knife cookbook create custom_mongo
```

in metadat.rb di custom_mongo add

```
depends          'docker', '~> 2.0'
```

Passaggio 3: crea attributo e ricetta

attributi

```
default['custom_mongo']['mongo_keyfile'] = '/data/keyfile'
default['custom_mongo']['mongo_datadir'] = '/data/db'
default['custom_mongo']['mongo_datapath'] = '/data'
default['custom_mongo']['keyfilename'] = 'mongodb-keyfile'
```

Ricetta

```
#
# Cookbook Name:: custom_mongo
# Recipe:: default
#
# Copyright 2017, Innocent Anigbo
#
# All rights reserved - Do Not Redistribute
#

data_path = "#{node['custom_mongo']['mongo_datapath']}"
data_dir = "#{node['custom_mongo']['mongo_datadir']}"
key_dir = "#{node['custom_mongo']['mongo_keyfile']}"
keyfile_content = data_bag_item('mongo-keyfile', 'keyfile')
keyfile_name = "#{node['custom_mongo']['keyfilename']}"

#chown of keyfile to docker user
execute 'assign-user' do
 command "chown 999 #{key_dir}/#{keyfile_name}"
 action :nothing
end

#Declaration to create Mongo data DIR and Keyfile DIR
%W[ #{data_path} #{data_dir} #{key_dir} ].each do |path|
directory path do
  mode '0755'
  end
end

#declaration to copy keyfile from data_bag to keyfile DIR on your mongo server
file "#{key_dir}/#{keyfile_name}" do
  content keyfile_content['key-file']
  group 'root'
  mode '0400'
  notifies :run, 'execute[assign-user]', :immediately
end

#Install docker
docker_service 'default' do
  action [:create, :start]
```

```
end

#Install mongo 3.4.2
docker_image 'mongo' do
  tag '3.4.2'
  action :pull
end
```

Crea ruolo chiamato mongo-role nella directory dei ruoli

```
{
  "name": "mongo-role",
  "description": "mongo DB Role",
  "run_list": [
    "recipe[custom_mongo]"
  ]
}
```

Aggiungi il ruolo in alto all'elenco dei tre nodi di mongo run

```
knife node run_list add FQDN_of_node_01 'role[mongo-role]'
knife node run_list add FQDN_of_node_02 'role[mongo-role]'
knife node run_list add FQDN_of_node_03 'role[mongo-role]'
```

Passaggio 4: inizializzare i tre nodi Mongo per formare un repset

Suppongo che il ruolo di cui sopra sia già stato applicato su tutti e tre i nodi Mongo. Solo sul nodo 01, Avvia Mongo con --auth per abilitare l'autenticazione

```
docker run --name mongo -v /data/db:/data/db -v /data/keyfile:/opt/keyfile --hostname="mongo-
01.example.com" -p 27017:27017 -d mongo:3.4.2 --keyFile /opt/keyfile/mongodb-keyfile --auth
```

Accedi alla shell interattiva del contenitore mobile in esecuzione sul nodo 01 e Crea utente amministratore

```
docker exec -it mongo /bin/sh
    mongo
    use admin
    db.createUser( {
         user: "admin-user",
         pwd: "password",
         roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
       });
```

Crea utente root

```
db.createUser( {
         user: "RootAdmin",
         pwd: "password",
         roles: [ { role: "root", db: "admin" } ]
       });
```

Arresta ed elimina il contenitore Docker creato sopra sul nodo 01. Ciò non influirà sui dati e sul file

di chiavi nell'host DIR. Dopo aver eliminato l'avvio di Mongo di nuovo sul nodo 01 ma questa volta con il flag di repset

```
docker rm -fv mongo
docker run --name mongo-uat -v /data/db:/data/db -v /data/keyfile:/opt/keyfile --
hostname="mongo-01.example.com" -p 27017:27017 -d mongo:3.4.2 --keyFile /opt/keyfile/mongodb-
keyfile --replSet "rs0"
```

ora avvia mongo sul nodo 02 e 03 con il flag di ripetizione

```
docker run --name mongo -v /data/db:/data/db -v /data/keyfile:/opt/keyfile --hostname="mongo-
02.example.com" -p 27017:27017 -d mongo:3.4.2 --keyFile /opt/keyfile/mongodb-keyfile --replSet
"rs0"
docker run --name mongo -v /data/db:/data/db -v /data/keyfile:/opt/keyfile --hostname="mongo-
03.example.com" -p 27017:27017 -d mongo:3.4.2 --keyFile /opt/keyfile/mongodb-keyfile --replSet
"rs0"
```

Autentica con l'utente root sul nodo 01 e avvia il set di repliche

```
use admin
db.auth("RootAdmin", "password");
rs.initiate()
```

Sul nodo 01 aggiungere Nodo 2 e 3 al Set di replica per formare il cluster repset0

```
rs.add("mongo-02.example.com")
rs.add("mongo-03.example.com")
```

analisi

Nell'esecuzione principale db.printSlaveReplicationInfo () e osservare SyncedTo e Behind the primary time. Il più tardi dovrebbe essere 0 sec come sotto

Produzione

```
rs0:PRIMARY> db.printSlaveReplicationInfo()
   source: mongo-02.example.com:27017
           syncedTo: Mon Mar 27 2017 15:01:04 GMT+0000 (UTC)
           0 secs (0 hrs) behind the primary
   source: mongo-03.example.com:27017
           syncedTo: Mon Mar 27 2017 15:01:04 GMT+0000 (UTC)
           0 secs (0 hrs) behind the primary
```

Spero che questo aiuti qualcuno

Leggi Come configurare la replica Mongo a tre nodi utilizzando l'immagine di Docker e Provisioned utilizzando Chef online: https://riptutorial.com/it/docker/topic/10014/come-configurare-la-replica-mongo-a-tre-nodi-utilizzando-l-immagine-di-docker-e-provisioned-utilizzando-chef

Capitolo 6: Come eseguire il debug quando la creazione della finestra mobile non riesce

introduzione

Quando un `docker build -t mytag .` fallisce con un messaggio come `---> Running in d9a42e53eb5a The command '/bin/sh -c returned a non-zero code: 127` (127 significa "comando non trovato, ma 1) non è banale per tutti 2) 127 può essere sostituito da 6 o qualsiasi cosa) può essere non banale per trovare l'errore in una lunga fila

Examples

esempio di base

Come l'ultimo livello creato da

```
docker build -t mytag .
```

ha mostrato

```
---> Running in d9a42e53eb5a
```

Basta lanciare l'ultima immagine creata con una shell e lanciare il comando, e si avrà un messaggio di errore più chiaro

```
docker run -it d9a42e53eb5a /bin/bash
```

(questo presuppone che / bin / bash sia disponibile, potrebbe essere / bin / sh o qualsiasi altra cosa)

e con il prompt, si lancia l'ultimo comando fallito e si vede ciò che viene visualizzato

Leggi Come eseguire il debug quando la creazione della finestra mobile non riesce online: https://riptutorial.com/it/docker/topic/8078/come-eseguire-il-debug-quando-la-creazione-della-finestra-mobile-non-riesce

Capitolo 7: Concetto di volumi Docker

Osservazioni

Le persone nuove in Docker spesso non si rendono conto che i filesystem Docker sono temporanei per impostazione predefinita. Se si avvia un'immagine Docker, si otterrà un contenitore che in superficie si comporta in modo molto simile a una macchina virtuale. È possibile creare, modificare ed eliminare file. Tuttavia, a differenza di una macchina virtuale, se interrompi il contenitore e lo avvii di nuovo, tutte le tue modifiche andranno perse: tutti i file che hai precedentemente eliminato torneranno e tutti i nuovi file o modifiche apportate non saranno presenti.

I volumi nei contenitori docker consentono dati persistenti e per la condivisione dei dati della macchina host all'interno di un contenitore.

Examples

A) Avviare un contenitore con un volume

```
[root@localhost ~]# docker run -it -v  /data  --name=vol3   8251da35e7a7 /bin/bash
root@d87bf9607836:/# cd /data/
root@d87bf9607836:/data# touch abc{1..10}
root@d87bf9607836:/data# ls
```

abc1 abc10 abc2 abc3 abc4 abc5 abc6 abc7 abc8 abc9

B) Ora premi [cont + P + Q] per uscire dal contenitore senza terminare il contenitore controllando il contenitore in esecuzione

```
[root@localhost ~]# docker ps
```

ID CONTENITORE IMMAGINE COMANDO CREATO STATO PORTO NOMI d87bf9607836 8251da35e7a7 "/ bin / bash" Circa un minuto fa Fino 31 secondi vol3 [root @ localhost ~] #

C) Esegui 'docker inspect' per controllare maggiori informazioni sul volume

```
[root@localhost ~]# docker inspect  d87bf9607836
```

"Supporti": [{"Nome": "cdf78fbf79a7c9363948e133abe4c572734cd788c95d36edea0448094ec9121c", "Origine": "/ var / lib / docker / volumi / cdf78fbf79a7c9363948e133abe4c572734cd788c95d36edea0448094ec9121c / _data", "Destinazione": "/ dati", "Driver": "locale", "Modalità": "", "RW": vero

D) È possibile allegare un volume di contenitori in esecuzione a un altro contenitore

```
[root@localhost ~]# docker run -it  --volumes-from  vol3  8251da35e7a7  /bin/bash

root@ef2f5cc545be:/# ls
```

bin boot data dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var

`root@ef2f5cc545be:/# ls` / data abc1 abc10 abc2 abc3 abc4 abc5 abc6 abc7 abc8 abc9

E) Puoi anche montare la tua directory di base all'interno del contenitore

```
[root@localhost ~]# docker run -it  -v  /etc:/etc1 8251da35e7a7 /bin/bash
```

Qui: / etc è la directory della macchina host e / etc1 è la destinazione all'interno del contenitore

Leggi Concetto di volumi Docker online: https://riptutorial.com/it/docker/topic/5908/concetto-di-volumi-docker

Capitolo 8: Costruire immagini

Parametri

Parametro	Dettagli
--Tirare	Garantisce che l'immagine di base (`FROM`) sia aggiornata prima di creare il resto del Dockerfile.

Examples

Costruire un'immagine da un Dockerfile

Una volta che hai un Dockerfile, puoi costruirne un'immagine usando la `docker build` . La forma base di questo comando è:

```
docker build -t image-name path
```

Se il tuo Dockerfile non è denominato `Dockerfile` , puoi usare il flag `-f` per dare il nome del Dockerfile da compilare.

```
docker build -t image-name -f Dockerfile2 .
```

Ad esempio, per creare un'immagine denominata `dockerbuild-example:1.0.0` da un `Dockerfile` nella directory di lavoro corrente:

```
$ ls
Dockerfile Dockerfile2

$ docker build -t dockerbuild-example:1.0.0 .

$ docker build -t dockerbuild-example-2:1.0.0 -f Dockerfile2 .
```

Vedere la documentazione di utilizzo della `docker build` per ulteriori opzioni e impostazioni.

Un errore comune è la creazione di un Dockerfile nella directory home dell'utente (`~`). Questa è una cattiva idea perché durante la `docker build -t mytag .` questo messaggio apparirà per molto tempo:

> Caricamento del contesto

La causa è il daemon docker che tenta di copiare tutti i file dell'utente (sia la directory home che le sottodirectory). Evita questo specificando sempre una directory per il Dockerfile.

Aggiungere un file `.dockerignore` alla directory di build è una buona pratica . La sua sintassi è simile ai file `.gitignore` e farà in modo che solo i file e le directory desiderati vengano caricati come

contesto del build.

Un semplice Dockerfile

```
FROM node:5
```

La direttiva `FROM` specifica un'immagine da cui iniziare. È possibile utilizzare qualsiasi riferimento di immagine valido.

```
WORKDIR /usr/src/app
```

La direttiva `WORKDIR` imposta la directory di lavoro corrente all'interno del contenitore, equivalente all'esecuzione di `cd` all'interno del contenitore. (Nota: il `RUN cd` *non* cambierà la directory di lavoro corrente).

```
RUN npm install cowsay knock-knock-jokes
```

`RUN` esegue il comando specificato all'interno del contenitore.

```
COPY cowsay-knockknock.js ./
```

`COPY` copia il file o la directory specificati nel primo argomento dal contesto di compilazione (il *`path`* passato al `docker build` *`path`*) nella posizione nel contenitore specificato dal secondo argomento.

```
CMD node cowsay-knockknock.js
```

`CMD` specifica un comando da eseguire quando viene eseguita l'immagine e non viene dato alcun comando. Può essere sovrascritto passando un comando alla `docker run` .

Ci sono molte altre istruzioni e opzioni; vedere il riferimento Dockerfile per un elenco completo.

Differenza tra ENTRYPOINT e CMD

Esistono due direttive `Dockerfile` per specificare quale comando eseguire per impostazione predefinita nelle immagini create. Se si specifica solo `CMD` finestra mobile eseguirà tale comando utilizzando il comando `ENTRYPOINT` predefinito, che è `/bin/sh -c` . È possibile sovrascrivere uno o entrambi il punto di accesso e / o il comando quando si avvia l'immagine costruita. Se si specificano entrambi, `ENTRYPOINT` specifica l'eseguibile del processo contenitore e `CMD` verrà fornito come parametri di tale eseguibile.

Ad esempio se il tuo `Dockerfile` contiene

```
FROM ubuntu:16.04
CMD ["/bin/date"]
```

Quindi si utilizza la direttiva `ENTRYPOINT` predefinita di `/bin/sh -c` e in esecuzione `/bin/date` con quel punto di accesso predefinito. Il comando del processo del contenitore sarà `/bin/sh -c /bin/date` .

Una volta eseguita questa immagine, per impostazione predefinita verrà stampata la data corrente

```
$ docker build -t test .
$ docker run test
Tue Jul 19 10:37:43 UTC 2016
```

È possibile eseguire l'override di `CMD` sulla riga di comando, nel qual caso verrà eseguito il comando specificato.

```
$ docker run test /bin/hostname
bf0274ec8820
```

Se si specifica una direttiva `ENTRYPOINT` , Docker utilizzerà quell'eseguibile e la direttiva `CMD` specifica i parametri predefiniti del comando. Quindi se il tuo `Dockerfile` contiene:

```
FROM ubuntu:16.04
ENTRYPOINT ["/bin/echo"]
CMD ["Hello"]
```

Quindi eseguirlo produrrà

```
$ docker build -t test .
$ docker run test
Hello
```

È possibile fornire diversi parametri se lo si desidera, ma verranno eseguiti tutti `/bin/echo`

```
$ docker run test Hi
Hi
```

Se si desidera sovrascrivere il punto di accesso elencato nel file Docker (ovvero se si desidera eseguire un comando diverso da `echo` in questo contenitore), è necessario specificare il parametro `--entrypoint` sulla riga di comando:

```
$ docker run --entrypoint=/bin/hostname test
b2c70e74df18
```

In genere si utilizza la direttiva `ENTRYPOINT` per puntare all'applicazione principale che si desidera eseguire e `CMD` ai parametri predefiniti.

Esporre una porta nel Dockerfile

```
EXPOSE <port> [<port>...]
```

Dalla documentazione di Docker:

> L'istruzione `EXPOSE` informa Docker che il contenitore è in ascolto sulle porte di rete specificate in fase di runtime. `EXPOSE` non rende accessibili le porte del contenitore all'host. Per fare ciò, è necessario utilizzare il flag `-p` per pubblicare un intervallo di

porte o il flag `-P` per pubblicare tutte le porte esposte. È possibile esporre un numero di porta e pubblicarlo esternamente con un altro numero.

Esempio:

Dentro il tuo Dockerfile:

```
EXPOSE 8765
```

Per accedere a questa porta dal computer host, includere questo argomento nel comando di `docker run` :

```
-p 8765:8765
```

ENTRYPOINT e CMD visti come verbo e parametro

Supponiamo di avere un Dockerfile che termina con

```
ENTRYPOINT [ "nethogs"] CMD ["wlan0"]
```

se costruisci questa immagine con a

```
docker built -t inspector .
```

lanciare l'immagine creata con un Dockerfile come tale con un comando come

```
docker run -it --net=host --rm inspector
```

, nethogs monitorerà l'interfaccia chiamata wlan0

Ora se vuoi monitorare l'interfaccia eth0 (o wlan1, o ra1 ...), farai qualcosa di simile

```
docker run -it --net=host --rm inspector eth0
```

o

```
docker run -it --net=host --rm inspector wlan1
```

Spingendo e tirando un'immagine nell'hub Docker o in un altro registro

Le immagini create localmente possono essere trasferite su Docker Hub o su qualsiasi altro host di repository docker, noto come registro. Utilizzare l' `docker login` per accedere a un account hub docker esistente.

```
docker login

Login with your Docker ID to push and pull images from Docker Hub.
If you don't have a Docker ID, head over to https://hub.docker.com to create one.

Username: cjsimon
Password:
```

```
Login Succeeded
```

Un diverso registro docker può essere utilizzato specificando un nome server. Questo funziona anche per i registri privati o self-hosted. Inoltre, è possibile utilizzare un archivio di credenziali esterne per la sicurezza.

```
docker login quay.io
```

È quindi possibile contrassegnare e inviare immagini al registro a cui si è effettuato l'accesso. Il tuo repository deve essere specificato come `server/username/reponame:tag` . L'omissione del server è attualmente impostata su Docker Hub. (Il registro predefinito non può essere cambiato con un altro provider e non ci sono piani per implementare questa funzione.)

```
docker tag mynginx quay.io/cjsimon/mynginx:latest
```

Tag diversi possono essere usati per rappresentare diverse versioni, o rami, della stessa immagine. Un'immagine con più tag diversi mostrerà ogni tag nello stesso repository.

Utilizza le `docker images` per visualizzare un elenco di immagini installate installate sul computer locale, inclusa l'immagine appena taggata. Quindi utilizzare push per caricarlo nel registro e tirare per scaricare l'immagine.

```
docker push quay.io/cjsimon/mynginx:latest
```

Tutti i tag di un'immagine possono essere estratti specificando l'opzione `-a`

```
docker pull quay.io/cjsimon/mynginx:latest
```

Costruire usando un proxy

Spesso durante la creazione di un'immagine Docker, il Dockerfile contiene istruzioni che eseguono programmi per recuperare risorse da Internet (per esempio `wget` per estrarre un build binario del programma su GitHub).

È possibile indicare a Docker di passare le variabili di ambiente set set in modo che tali programmi eseguano tali recuperi attraverso un proxy:

```
$ docker build --build-arg http_proxy=http://myproxy.example.com:3128 \
               --build-arg https_proxy=http://myproxy.example.com:3128 \
               --build-arg no_proxy=internal.example.com \
               -t test .
```

`build-arg` sono variabili d'ambiente disponibili solo al momento della compilazione.

Leggi Costruire immagini online: https://riptutorial.com/it/docker/topic/713/costruire-immagini

Capitolo 9: Creare un servizio con persistenza

Sintassi

- volume della finestra mobile create --name <volume_name> # Crea un volume chiamato <nome_volume>
- finestra mobile run -v <volume_name>: <punto di mount> -d crramirez / limesurvey: latest # Montare il volume <volume_name> nella directory <mount_point> nel contenitore

Parametri

Parametro	Dettagli
--name <volume_name>	Specificare il nome del volume da creare
-v <nome_volume>: <punto_montaggio>	Specificare dove verrà montato il volume denominato nel contenitore

Osservazioni

La persistenza viene creata nei contenitori mobili utilizzando i volumi. Docker ha molti modi per gestire i volumi. I volumi denominati sono molto convenienti da:

- Persistono anche quando il contenitore viene rimosso usando l'opzione -v.
- L'unico modo per eliminare un volume denominato è eseguire una chiamata esplicita al volume della finestra mobile rm
- I volumi nominati possono essere condivisi tra container senza collegamento o opzione --volumes-from.
- Non hanno problemi di autorizzazione che hanno i volumi montati dall'host.
- Possono essere manipolati utilizzando il comando del volume della finestra mobile.

Examples

Persistenza con volumi denominati

La persistenza viene creata nei contenitori mobili utilizzando i volumi. Creiamo un container Limesurvey e manteniamo il database, il contenuto caricato e la configurazione in un volume denominato:

```
docker volume create --name mysql
docker volume create --name upload
```

```
docker run -d --name limesurvey -v mysql:/var/lib/mysql -v upload:/app/upload -p 80:80
crramirez/limesurvey:latest
```

Backup di un contenuto del volume con nome

Dobbiamo creare un contenitore per montare il volume. Quindi archivialo e scarica l'archivio sul nostro host.

Creiamo prima un volume di dati con alcuni dati:

```
docker volume create --name=data
echo "Hello World" |  docker run -i --rm=true -v data:/data ubuntu:trusty tee /data/hello.txt
```

Eseguiamo il backup dei dati:

```
docker run -d --name backup -v data:/data ubuntu:trusty tar -czvf /tmp/data.tgz /data
docker cp backup:/tmp/data.tgz data.tgz
docker rm -fv backup
```

Proviamo:

```
tar -xzvf data.tgz
cat data/hello.txt
```

Leggi Creare un servizio con persistenza online: https://riptutorial.com/it/docker/topic/7429/creare-un-servizio-con-persistenza

Capitolo 10: Debug di un contenitore

Sintassi

- statistiche docker [OPZIONI] [CONTENITORE ...]
- log del docker [OPZIONI] CONTENITORE
- finestra mobile [OPZIONI] CONTENITORE [ps OPZIONI]

Examples

Entrare in un contenitore funzionante

Per eseguire operazioni in un contenitore, utilizzare il comando `docker exec` . A volte questo è chiamato "entrare nel contenitore" poiché tutti i comandi sono eseguiti all'interno del contenitore.

```
docker exec -it container_id bash
```

o

```
docker exec -it container_id /bin/sh
```

E ora hai una shell nel tuo contenitore funzionante. Ad esempio, elenca i file in una directory e lascia il contenitore:

```
docker exec container_id ls -la
```

È possibile utilizzare il `-u flag` per immettere il contenitore con un utente specifico, ad esempio, `uid=1013` , `gid=1023` .

```
docker exec -it -u 1013:1023 container_id ls -la
```

L'uid e il gid non devono esistere nel contenitore ma il comando può causare errori. Se si desidera avviare un contenitore e accedere immediatamente al fine di controllare qualcosa, è possibile farlo

```
docker run...; docker exec -it $(docker ps -lq) bash
```

il comando `docker ps -lq` restituisce solo l'id dell'ultimo (il contenitore l in `-lq`) avviato. (questo suppone che tu abbia bash come interprete disponibile nel tuo contenitore, potresti avere sh o zsh o qualsiasi altro)

Monitorare l'utilizzo delle risorse

Ispezionare l'utilizzo delle risorse di sistema è un modo efficace per trovare applicazioni che presentano un comportamento anomalo. Questo esempio è equivalente al comando `top` tradizionale per i contenitori:

```
docker stats
```

Per seguire le statistiche di contenitori specifici, elencali sulla riga di comando:

```
docker stats 7786807d8084 7786807d8085
```

Le statistiche Docker mostrano le seguenti informazioni:

```
CONTAINER     CPU %   MEM USAGE / LIMIT   MEM %    NET I/O               BLOCK I/O
7786807d8084  0.65%   1.33 GB / 3.95 GB   33.67%   142.2 MB / 57.79 MB   46.32 MB / 0 B
```

Per impostazione predefinita, le `docker stats` visualizzano l'id dei contenitori, e questo non è molto utile, se preferisci visualizzare i nomi del container, basta fare

```
docker stats $(docker ps --format '{{.Names}}')
```

Monitoraggio dei processi in un contenitore

Ispezionare l'utilizzo delle risorse di sistema è un modo efficace per limitare un problema su un'applicazione live. Questo esempio è un equivalente del comando tradizionale `ps` per contenitori.

```
docker top 7786807d8084
```

Per filtrare il formato dell'output, aggiungi le opzioni `ps` sulla riga di comando:

```
docker top 7786807d8084 faux
```

Oppure, per ottenere l'elenco dei processi in esecuzione come root, che è una pratica potenzialmente dannosa:

```
docker top 7786807d8084 -u root
```

Il comando `docker top` risulta particolarmente utile quando si risolvono i problemi di contenitori minimalistici senza shell o il comando `ps` .

Allegare a un contenitore in esecuzione

'Allegare a un contenitore' è l'atto di avviare una sessione terminale all'interno del contesto in cui è in esecuzione il contenitore (e tutti i programmi in esso contenuti). Questo è principalmente usato per scopi di debug, ma potrebbe anche essere necessario se dati specifici devono essere passati a programmi in esecuzione all'interno del contenitore.

Il comando `attach` è utilizzato per fare questo. Ha questa sintassi:

```
docker attach <container>
```

`<container>` può essere l'id del contenitore o il nome del contenitore. Per esempio:

```
docker attach c8a9cf1a1fa8
```

O:

```
docker attach graceful_hopper
```

Potrebbe essere necessario `sudo` i comandi di cui sopra, a seconda del vostro utente e come finestra mobile è impostato.

> Nota: Collega consente solo una sessione di shell singola da collegare a un contenitore alla volta.
>
> Attenzione: *tutti gli* input da tastiera verranno inoltrati al contenitore. Colpire `Ctrl-c` *ucciderà il* tuo contenitore.

Per staccare da un contenitore collegato, premere successivamente `Ctrl-p e` poi `Ctrl-q`

Per collegare più sessioni di shell a un contenitore o semplicemente come alternativa, è possibile utilizzare `exec` . Utilizzando l'id del contenitore:

```
docker exec -i -t c8a9cf1a1fa8 /bin/bash
```

Usando il nome del contenitore:

```
docker exec -i -t graceful_hopper /bin/bash
```

`exec` eseguirà un programma all'interno di un contenitore, in questo caso `/bin/bash` (una shell, presumibilmente una che ha il contenitore). `-i` indica una sessione interattiva, mentre `-t` alloca uno pseudo-TTY.

> Nota: Diversamente dal *collegamento* , premendo `Ctrl-c` si termina il comando *exec* 'd solo quando viene eseguito in modo interattivo.

Stampa dei registri

Seguire i registri è il modo meno intrusivo per eseguire il debug di un'applicazione live. Questo esempio riproduce il comportamento della `tail -f some-application.log` tradizionale `tail -f some-application.log` sul contenitore `7786807d8084` .

```
docker logs --follow --tail 10 7786807d8084
```

Questo comando mostra fondamentalmente l'output standard del processo contenitore (il processo con pid 1).

Se i tuoi log non includono in modo nativo il timestamping, puoi aggiungere il flag `--timestamps` .

È possibile anche guardare i log di un container fermo

- avvia il contenitore `docker run ... ; docker logs $(docker ps -lq)` con la `docker run ... ; docker logs $(docker ps -lq)`
- trova l'ID contenitore o il nome con

```
docker ps -a
```

e poi

```
docker logs container-id
```

o

```
docker logs containername
```

come è possibile guardare i registri di un container fermo

Debugging del processo contenitore Docker

Docker è solo un modo elegante per eseguire un processo, non una macchina virtuale. Pertanto, il debug di un processo "in un contenitore" è anche possibile "sull'host" semplicemente esaminando il processo del contenitore in esecuzione come utente con le autorizzazioni appropriate per ispezionare quei processi sull'host (ad es. Root). Ad esempio, è possibile elencare ogni "processo contenitore" sull'host eseguendo un semplice `ps` come root:

```
sudo ps aux
```

Tutti i contenitori Docker attualmente in esecuzione saranno elencati nell'output.

Questo può essere utile durante lo sviluppo dell'applicazione per il debug di un processo in esecuzione in un contenitore. Come utente con autorizzazioni appropriate, è possibile utilizzare utility di debug tipiche nel processo contenitore, come strace, ltrace, gdb, ecc.

Leggi Debug di un contenitore online: https://riptutorial.com/it/docker/topic/1333/debug-di-un-contenitore

Capitolo 11: Docker in Docker

Examples

Contenitore CI Jenkins che utilizza Docker

Questo capitolo descrive come configurare un Docker Container con Jenkins all'interno, che è in grado di inviare comandi Docker all'installazione Docker (il daemon Docker) dell'host. Utilizzare in modo efficace Docker in Docker. Per raggiungere questo obiettivo, dobbiamo creare un'immagine Docker personalizzata basata su una versione arbitraria dell'immagine ufficiale di Docker di Jenkins. Il Dockerfile (l'istruzione come costruire l'immagine) assomiglia a questo:

```
FROM jenkins

USER root

RUN cd /usr/local/bin && \
curl https://master.dockerproject.org/linux/amd64/docker > docker  && \
chmod +x docker  && \
groupadd -g 999 docker && \
usermod -a -G docker jenkins

USER Jenkins
```

Questo Dockerfile crea un'immagine che contiene i binari del client Docker utilizzato da questo client per comunicare con un daemon Docker. In questo caso il Docker Daemon dell'host. L'istruzione `RUN` in questo file crea anche un gruppo di utenti UNIX con l'UID 999 e aggiunge l'utente Jenkins. Perché esattamente questo è necessario è descritto nel capitolo successivo. Con questa immagine possiamo eseguire un server Jenkins che può usare i comandi Docker, ma se eseguiamo questa immagine il client Docker che abbiamo installato all'interno dell'immagine non può comunicare con il daemon Docker dell'host. Questi due componenti comunicano tramite UNIX Socket `/var/run/docker.sock` . Su Unix questo è un file come tutto il resto, quindi possiamo montarlo facilmente all'interno del contenitore Jenkins. Questo viene fatto con il comando `docker run -v /var/run/docker.sock:/var/run/docker.sock --name jenkins MY_CUSTOM_IMAGE_NAME` . Ma questo file montato è di proprietà di `docker:root` e per questo motivo il Dockerfile crea questo gruppo con un UID ben noto e aggiunge l'utente Jenkins ad esso. Ora il contenitore Jenkins è davvero in grado di funzionare e utilizzare Docker. In produzione, il comando run dovrebbe contenere anche `-v jenkins_home:/var/jenkins_home` per `-v jenkins_home:/var/jenkins_home` backup della directory Jenkins_home e, naturalmente, una mappatura delle porte per accedere al server sulla rete.

Leggi Docker in Docker online: https://riptutorial.com/it/docker/topic/8012/docker-in-docker

Capitolo 12: Docker Machine

introduzione

Gestione remota di più host del motore mobile.

Osservazioni

`docker-machine` gestisce gli host remoti che eseguono Docker.

Lo strumento da riga di comando della `docker-machine` consente di gestire l'intero ciclo di vita della macchina utilizzando driver specifici del provider. Può essere utilizzato per selezionare una macchina "attiva". Una volta selezionato, una macchina attiva può essere utilizzata come se fosse il motore Docker locale.

Examples

Ottieni informazioni aggiornate sull'ambiente di Docker Machine

Tutti questi sono comandi della shell.

`docker-machine env` per ottenere la configurazione corrente della macchina docker predefinita

`eval $(docker-machine env)` per ottenere la configurazione corrente della macchina docker e impostare l'ambiente shell corrente su per utilizzare questa macchina docker.

Se la tua shell è impostata per usare un proxy, puoi specificare l'opzione --no-proxy per bypassare il proxy quando ti connetti alla tua macchina mobile: `eval $(docker-machine env --no-proxy)`

Se si hanno più macchine docker, è possibile specificare il nome macchina come argomento: `eval $(docker-machine env --no-proxy machinename)`

SSH in una finestra mobile

Tutti questi sono comandi della shell

- Se è necessario accedere direttamente a una finestra mobile in esecuzione, è possibile farlo:

`docker-machine ssh` per ssh nella finestra mobile predefinita

`docker-machine ssh machinename` per ssh in una finestra mobile non predefinita

- Se vuoi solo eseguire un singolo comando, puoi farlo. Per eseguire il `uptime` sulla finestra mobile predefinita per vedere per quanto tempo è in esecuzione, eseguire `docker-machine ssh default uptime`

Crea una macchina Docker

L'uso di `docker-machine` è il metodo migliore per installare Docker su una macchina. Applicherà automaticamente le migliori impostazioni di sicurezza disponibili, compresa la generazione di una coppia unica di certificati SSL per l'autenticazione reciproca e le chiavi SSH.

Per creare una macchina locale utilizzando Virtualbox:

```
docker-machine create --driver virtualbox docker-host-1
```

Per installare Docker su una macchina esistente, utilizzare il driver `generic` :

```
docker-machine -D create -d generic --generic-ip-address 1.2.3.4 docker-host-2
```

L'opzione `--driver` dice a `--driver` come creare la macchina. Per un elenco dei driver supportati, vedere:

- ufficialmente supportato
- terzo

Elenca le macchine mobili

Elenco delle macchine docker restituirà lo stato, l'indirizzo e la versione di Docker di ciascuna macchina mobile.

```
 docker-machine ls
```

Stamperà qualcosa come:

```
NAME             ACTIVE   DRIVER    STATE     URL                         SWARM   DOCKER
ERRORS
docker-machine-1 -        ovh       Running   tcp://1.2.3.4:2376                  v1.11.2
docker-machine-2 -        generic   Running   tcp://1.2.3.5:2376                  v1.11.2
```

Per elencare le macchine in esecuzione:

```
docker-machine ls --filter state=running
```

Per elencare le macchine degli errori:

```
docker-machine ls --filter state=
```

Per elencare le macchine il cui nome inizia con "side-project-", usa il filtro Golang:

```
docker-machine ls --filter name="^side-project-"
```

Per ottenere solo l'elenco degli URL della macchina:

```
docker-machine ls --format '{{ .URL }}'
```

Vedere https://docs.docker.com/machine/reference/ls/ per il riferimento completo del comando.

Aggiorna una finestra mobile

L'aggiornamento di una finestra mobile implica un periodo di inattività e potrebbe richiedere la piallatura. Per aggiornare una finestra mobile, eseguire:

```
docker-machine upgrade docker-machine-name
```

Questo comando non ha opzioni

Ottieni l'indirizzo IP di una finestra mobile

Per ottenere l'indirizzo IP di una finestra mobile, puoi farlo con questo comando:

```
docker-machine ip machine-name
```

Leggi Docker Machine online: https://riptutorial.com/it/docker/topic/1349/docker-machine

Capitolo 13: Docker network

Examples

Come trovare l'IP dell'host del contenitore

È necessario trovare l'indirizzo IP del contenitore in esecuzione nell'host in modo che sia possibile, ad esempio, connettersi al server Web in esecuzione.

`docker-machine` è ciò che viene utilizzato su MacOSX e Windows.

In primo luogo, elenca le tue macchine:

```
$ docker-machine ls

NAME      ACTIVE   DRIVER       STATE     URL                         SWARM
default   *        virtualbox   Running   tcp://192.168.99.100:2376
```

Quindi selezionare una delle macchine (quella predefinita è definita predefinita) e:

```
$ docker-machine ip default

192.168.99.100
```

Creazione di una rete Docker

```
docker network create app-backend
```

Questo comando creerà una semplice rete a ponte chiamata `appBackend` . Nessun contenitore è collegato a questa rete per impostazione predefinita.

Elenco delle reti

```
docker network ls
```

Questo comando elenca tutte le reti che sono state create sull'host Docker locale. Include la rete `bridge` bridge predefinita, la rete host `host` e la rete null `null` . Tutti i contenitori per impostazione predefinita sono collegati alla rete `bridge` bridge predefinita.

Aggiungi contenitore alla rete

```
docker network connect app-backend myAwesomeApp-1
```

Questo comando collega il contenitore `myAwesomeApp-1` alla rete di `app-backend` . Quando si aggiunge un contenitore a una rete definita dall'utente, il resolver DNS incorporato (che non è un server

DNS completo e non esportabile) consente a ciascun contenitore sulla rete di risolvere l'altro contenitore sulla stessa rete. Questo semplice resolver DNS non è disponibile sulla rete `bridge` bridge predefinita.

Scollegare il contenitore dalla rete

```
docker network disconnect app-backend myAwesomeApp-1
```

Questo comando scollega il contenitore `myAwesomeApp-1` dalla rete di `app-backend` . Il contenitore non sarà più in grado di comunicare con altri contenitori sulla rete da cui è stato disconnesso, né utilizzare il resolver DNS incorporato per cercare altri contenitori sulla rete da cui è stato rimosso.

Rimuovere una rete Docker

```
docker network rm app-backend
```

Questo comando rimuove la rete di `app-backend` definita dall'utente dall'host Docker. Tutti i contenitori della rete non collegati in altro modo tramite un'altra rete perderanno la comunicazione con altri contenitori. Non è possibile rimuovere la rete `bridge` bridge predefinita, la rete `host` host o la rete `null` null.

Ispeziona una rete Docker

```
docker network inspect app-backend
```

Questo comando mostrerà i dettagli sulla rete di `app-backend` .

L'output di questo comando dovrebbe essere simile a:

```
[
    {
        "Name": "foo",
        "Id": "a0349d78c8fd7c16f5940bdbaf1adec8d8399b8309b2e8a969bd4e3226a6fc58",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1/16"
                }
            ]
        },
        "Internal": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
```

```
]
```

Leggi Docker network online: https://riptutorial.com/it/docker/topic/3221/docker-network

Capitolo 14: Dockerfiles

introduzione

I file Docker sono file utilizzati per creare in modo programmatico immagini Docker. Ti consentono di creare in modo rapido e riproducibile un'immagine Docker e quindi sono utili per la collaborazione. I file Docker contengono istruzioni per la creazione di un'immagine Docker. Ogni istruzione è scritta su una riga e viene data nella forma `<INSTRUCTION><argument(s)>` . I Dockerfile sono usati per costruire immagini Docker usando il comando di `docker build` .

Osservazioni

I Dockerfiles sono nella forma:

```
# This is a comment
INSTRUCTION arguments
```

- I commenti iniziano con `#`
- Le istruzioni sono solo in maiuscolo
- La prima istruzione di un Dockerfile deve essere `FROM` per specificare l'immagine di base

---

Durante la creazione di un Dockerfile, il client Docker invierà un "build context" al daemon Docker. Il contesto di costruzione include tutti i file e le cartelle nella stessa directory del file Docker. `COPY` operazioni `COPY` e `ADD` possono solo utilizzare file da questo contesto.

Alcuni file Docker possono iniziare con:

```
# escape=`
```

Questo è usato per istruire il parser Docker per usare `` ` `` come carattere di escape invece di `\` . Questo è utile soprattutto per i file di Windows Docker.

Examples

HelloWorld Dockerfile

Un Dockerfile minimale si presenta così:

```
FROM alpine
CMD ["echo", "Hello StackOverflow!"]
```

Ciò istruirà Docker per creare un'immagine basata su Alpine (`FROM`), una distribuzione minima per i contenitori e per eseguire un comando specifico (`CMD`) quando si esegue l'immagine risultante.

Costruisci ed eseguilo:

```
docker build -t hello .
docker run --rm hello
```

Questo produrrà:

```
Hello StackOverflow!
```

Copia di file

Per copiare file dal contesto di costruzione in un'immagine Docker, usa l'istruzione `COPY` :

```
COPY localfile.txt containerfile.txt
```

Se il nome del file contiene spazi, utilizza la sintassi alternativa:

```
COPY ["local file", "container file"]
```

Il comando `COPY` supporta i caratteri jolly. Può essere usato ad esempio per copiare tutte le immagini nella cartella `images/` :

```
COPY *.jpg images/
```

Nota: in questo esempio, le `images/` potrebbero non esistere. In questo caso, Docker lo creerà automaticamente.

Esporre una porta

Dichiarare porte esposte da un Dockerfile utilizzare la `EXPOSE` istruzione:

```
EXPOSE 8080 8082
```

L'impostazione delle porte esposte può essere sovrascritta dalla riga di comando Docker, ma è buona norma impostarle in modo esplicito nel Dockerfile in quanto aiuta a capire cosa fa un'applicazione.

Dockerfiles migliori pratiche

Raggruppare le operazioni comuni

Docker crea immagini come una raccolta di livelli. Ogni livello può solo aggiungere dati, anche se questi dati indicano che un file è stato cancellato. Ogni istruzione crea un nuovo livello. Per esempio:

```
RUN apt-get -qq update
RUN apt-get -qq install some-package
```

Ha un paio di aspetti negativi:

- Creerà due livelli, producendo un'immagine più grande.
- L'utilizzo `apt-get update` da solo in un'istruzione `RUN` causa problemi di memorizzazione nella cache e successivamente le istruzioni di `apt-get install` potrebbero **non riuscire** . Si supponga di modificare in seguito `apt-get install` aggiungendo pacchetti aggiuntivi, quindi la finestra mobile interpreta le istruzioni iniziali e modificate come identiche e riutilizza la cache dai passaggi precedenti. Di conseguenza il comando `apt-get update` **non** viene eseguito perché la sua versione cache viene utilizzata durante la compilazione.

Invece, usa:

```
RUN apt-get -qq update && \
    apt-get -qq install some-package
```

come questo produce solo un livello.

Cita il manutentore

Questa è solitamente la seconda riga del Dockerfile. Indica chi è responsabile e sarà in grado di aiutare.

```
LABEL maintainer John Doe <john.doe@example.com>
```

Se lo salti, non romperà la tua immagine. Ma non aiuterà neanche i tuoi utenti.

Sii conciso

Tieni corto il tuo Dockerfile. Se è necessaria una configurazione complessa, prendere in considerazione l'utilizzo di uno script dedicato o l'impostazione di immagini di base.

Istruzioni per l'utente

```
USER daemon
```

L'istruzione `USER` imposta il nome utente o l'UID da utilizzare quando si esegue l'immagine e per tutte le istruzioni `RUN` , `CMD` e `ENTRYPOINT` che seguono nel file `Dockerfile` .

Istruzione WORKDIR

```
WORKDIR /path/to/workdir
```

L'istruzione `WORKDIR` imposta la directory di lavoro per tutte le istruzioni `RUN` , `CMD` , `ENTRYPOINT` , `COPY` e `ADD` che seguono nel Dockerfile. Se il `WORKDIR` non esiste, verrà creato anche se non viene utilizzato in alcuna istruzione `Dockerfile` successiva.

Può essere utilizzato più volte nell'unico file `Dockerfile` . Se viene fornito un percorso relativo, sarà relativo al percorso della precedente istruzione `WORKDIR` . Per esempio:

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

L'output del comando `pwd` finale in questo `Dockerfile` sarebbe `/a/b/c` .

L'istruzione `WORKDIR` può risolvere le variabili di ambiente precedentemente impostate tramite `ENV` . È possibile utilizzare solo le variabili di ambiente impostate in modo esplicito nel file `Dockerfile` . Per esempio:

```
ENV DIRPATH /path
WORKDIR $DIRPATH/$DIRNAME
RUN pwd
```

L'output del comando `pwd` finale in questo Dockerfile sarebbe `/path/$DIRNAME`

VOLUME Istruzione

```
VOLUME ["/data"]
```

L'istruzione `VOLUME` crea un punto di montaggio con il nome specificato e lo contrassegna come contenente volumi montati esternamente dall'host nativo o da altri contenitori. Il valore può essere un array JSON, `VOLUME ["/var/log/"]` o una stringa semplice con più argomenti, come `VOLUME /var/log` o `VOLUME /var/log /var/db` . Per ulteriori informazioni / esempi e istruzioni di montaggio tramite il client Docker, fare riferimento a Condividi directory tramite la documentazione Volumi.

Il comando di `docker run` inizializza il volume appena creato con tutti i dati esistenti nella posizione specificata all'interno dell'immagine di base. Ad esempio, considera il seguente snippet Dockerfile:

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
```

Questo file Docker produce un'immagine che fa girare la finestra mobile, per creare un nuovo punto di montaggio su / myvol e copiare il file di benvenuto nel volume appena creato.

Nota: se alcuni passi di costruzione modificano i dati all'interno del volume dopo che è stato dichiarato, tali modifiche verranno scartate.

Nota: l'elenco viene analizzato come un array JSON, il che significa che è necessario utilizzare virgolette (") attorno a words not single-quotes (').

Istruzione di COPY

`COPY` ha due forme:

```
COPY <src>... <dest>
```

```
COPY ["<src>",... "<dest>"] (this form is required for paths containing whitespace)
```

L'istruzione `COPY` copia nuovi file o directory da `<src>` e li aggiunge al filesystem del contenitore nel percorso `<dest>` .

È possibile specificare più risorse `<src>` ma devono essere relative alla directory di origine che viene creata (il contesto della generazione).

Ogni `<src>` può contenere caratteri jolly e la corrispondenza verrà effettuata utilizzando le regole del `filepath.Match` di Go. Per esempio:

```
COPY hom* /mydir/        # adds all files starting with "hom"
COPY hom?.txt /mydir/    # ? is replaced with any single character, e.g., "home.txt"
```

`<dest>` è un percorso assoluto o un percorso relativo a `WORKDIR` , in cui la sorgente verrà copiata all'interno del contenitore di destinazione.

```
COPY test relativeDir/   # adds "test" to `WORKDIR`/relativeDir/
COPY test /absoluteDir/  # adds "test" to /absoluteDir/
```

Tutti i nuovi file e directory vengono creati con un UID e GID di 0.

Nota: se si crea usando stdin (`docker build - < somefile`), non esiste un contesto di build, quindi non è possibile utilizzare `COPY` .

`COPY` rispetta le seguenti regole:

- Il percorso `<src>` deve essere all'interno del contesto della build; non è possibile `COPY` ../qualcosa / qualcosa, poiché il primo passaggio di una build finestra mobile consiste nell'inviare la directory di contesto (e le sottodirectory) al daemon docker.
- Se `<src>` è una directory, vengono copiati tutti i contenuti della directory, inclusi i metadati del filesystem. Nota: la directory non viene copiata, ma solo il suo contenuto.
- Se `<src>` è un qualsiasi altro tipo di file, viene copiato individualmente insieme ai suoi metadati. In questo caso, se `<dest>` termina con una barra finale, verrà considerata una directory e il contenuto di `<src>` verrà scritto in `<dest>/base(<src>)` .
- Se vengono specificate più risorse `<src>` , direttamente o a causa dell'uso di un carattere jolly, allora `<dest>` deve essere una directory e deve terminare con una barra `/` .
- Se `<dest>` non termina con una barra finale, verrà considerato un file normale e il contenuto di `<src>` verrà scritto in `<dest>` .
- Se `<dest>` non esiste, viene creato insieme a tutte le directory mancanti nel suo percorso.

Le istruzioni ENV e ARG

ENV

```
ENV <key> <value>
ENV <key>=<value> ...
```

L'istruzione `ENV` imposta la variabile di ambiente `<key>` sul valore. Questo valore sarà nell'ambiente di tutti i comandi "discendenti" Dockerfile e può essere sostituito anche in linea in molti.

L'istruzione `ENV` ha due forme. Il primo modulo, `ENV <key> <value>` , imposterà una singola variabile su un valore. L'intera stringa dopo il primo spazio verrà considerata come `<value>` , inclusi caratteri come spazi e virgolette.

Il secondo modulo, `ENV <key>=<value> ...` , consente di impostare più variabili contemporaneamente. Si noti che il secondo modulo utilizza il segno di uguale (=) nella sintassi, mentre il primo modulo non lo fa. Come l'analisi della riga di comando, le virgolette e le barre retroverse possono essere utilizzate per includere spazi all'interno dei valori.

Per esempio:

```
ENV myName="John Doe" myDog=Rex\ The\ Dog \
    myCat=fluffy
```

e

```
ENV myName John Doe
ENV myDog Rex The Dog
ENV myCat fluffy
```

produrrà gli stessi risultati netti nel contenitore finale, ma la prima forma è preferita perché produce un singolo livello di cache.

Le variabili di ambiente impostate utilizzando `ENV` permangono quando viene eseguito un contenitore dall'immagine risultante. È possibile visualizzare i valori utilizzando il controllo finestra mobile e modificarli utilizzando la `docker run --env <key>=<value>` .

ARG

Se non desideri mantenere l'impostazione, usa invece `ARG` . `ARG` imposterà gli ambienti solo durante la compilazione. Ad esempio, impostazione

```
ENV DEBIAN_FRONTEND noninteractive
```

potrebbe confondere gli utenti `apt-get` su un'immagine basata su Debian quando entrano nel contenitore in un contesto interattivo tramite `docker exec -it the-container bash` .

Invece, usa:

```
ARG DEBIAN_FRONTEND noninteractive
```

In alternativa, puoi anche impostare un valore per un singolo comando usando solo:

```
RUN <key>=<value> <command>
```

ESPORTAZIONE

```
EXPOSE <port> [<port>...]
```

L'istruzione `EXPOSE` informa Docker che il contenitore è in ascolto sulle porte di rete specificate in fase di runtime. `EXPOSE` NON rende accessibili le porte del contenitore all'host. Per fare ciò, è necessario utilizzare il flag `-p` per pubblicare un intervallo di porte o il flag `-P` per pubblicare tutte le porte esposte. Questi flag vengono utilizzati nella `docker run [OPTIONS] IMAGE [COMMAND][ARG...]` per esporre la porta all'host. È possibile esporre un numero di porta e pubblicarlo esternamente con un altro numero.

```
docker run -p 2500:80 <image name>
```

Questo comando creerà un contenitore con il nome <image> e rilegherà la porta del contenitore 80 alla porta 2500 della macchina host.

Per impostare il reindirizzamento della porta sul sistema host, vedere l'uso `-P` . La funzione di rete Docker supporta la creazione di reti senza la necessità di esporre le porte all'interno della rete, per informazioni dettagliate consultare la panoramica di questa funzionalità).

Istruzione LABEL

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

L'istruzione `LABEL` aggiunge metadati a un'immagine. A `LABEL` è una coppia chiave-valore. Per includere spazi all'interno di un valore `LABEL` , utilizza virgolette e barre retroverse come faresti nell'analisi della riga di comando. Alcuni esempi di utilizzo:

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

Un'immagine può avere più di un'etichetta. Per specificare più etichette, Docker consiglia di combinare le etichette in un'unica istruzione `LABEL` laddove possibile. Ogni istruzione `LABEL` produce un nuovo livello che può causare un'immagine inefficiente se si utilizzano molte etichette. Questo esempio si traduce in un singolo livello di immagine.

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

Quanto sopra può anche essere scritto come:

```
LABEL multi.label1="value1" \
      multi.label2="value2" \
```

```
      other="value3"
```

Le etichette sono additive incluse le `LABEL` nelle immagini `FROM` . Se Docker rileva un'etichetta / chiave già esistente, il nuovo valore sostituisce tutte le etichette precedenti con chiavi identiche.

Per visualizzare le etichette di un'immagine, utilizzare il comando di controllo finestra mobile.

```
"Labels": {
    "com.example.vendor": "ACME Incorporated"
    "com.example.label-with-value": "foo",
    "version": "1.0",
    "description": "This text illustrates that label-values can span multiple lines.",
    "multi.label1": "value1",
    "multi.label2": "value2",
    "other": "value3"
},
```

Istruzione CMD

L'istruzione `CMD` ha tre forme:

```
CMD ["executable","param1","param2"] (exec form, this is the preferred form)
CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
CMD command param1 param2 (shell form)
```

Ci può essere solo un'istruzione `CMD` in un `Dockerfile` . Se si elencano più di un `CMD` allora solo l'ultimo `CMD` avrà effetto.

Lo scopo principale di un `CMD` è di fornire i valori predefiniti per un contenitore in esecuzione. Questi valori predefiniti possono includere un eseguibile oppure possono omettere l'eseguibile, nel qual caso è necessario specificare anche un'istruzione `ENTRYPOINT` .

Nota: se `CMD` viene utilizzato per fornire argomenti predefiniti per l'istruzione `ENTRYPOINT` , entrambe le istruzioni `CMD` e `ENTRYPOINT` devono essere specificate con il formato di array JSON.

Nota: il modulo exec viene analizzato come un array JSON, il che significa che è necessario utilizzare virgolette (") attorno a words not single-quotes (').

Nota: a differenza del modulo shell, il modulo exec non richiama una shell di comando. Ciò significa che la normale elaborazione della shell non avviene. Ad esempio, `CMD [ "echo", "$HOME" ]` non eseguirà la sostituzione delle variabili su `$HOME` . Se si desidera l'elaborazione della shell, utilizzare la forma della shell o eseguire direttamente una shell, ad esempio: `CMD [ "sh", "-c", "echo $HOME" ]` .

Quando viene utilizzato nei formati shell o exec, l'istruzione `CMD` imposta il comando da eseguire quando si esegue l'immagine.

Se si utilizza il modulo shell della `CMD` , il comando verrà eseguito in `/bin/sh -c` :

```
FROM ubuntu
```

```
CMD echo "This is a test." | wc -
```

Se si desidera eseguire il comando senza shell, è necessario esprimere il comando come array JSON e fornire il percorso completo dell'eseguibile. Questa forma di matrice è il formato preferito di `CMD` . Eventuali parametri aggiuntivi devono essere espressi singolarmente come stringhe nell'array:

```
FROM ubuntu
CMD ["/usr/bin/wc","--help"]
```

Se si desidera che il contenitore esegua sempre lo stesso eseguibile, è consigliabile utilizzare `ENTRYPOINT` in combinazione con `CMD` . Vedi `ENTRYPOINT` .

Se l'utente specifica gli argomenti sulla finestra mobile, questi sovrascriveranno il valore predefinito specificato in `CMD` .

Nota: non confondere `RUN` con `CMD` . `RUN` esegue effettivamente un comando al momento della creazione dell'immagine e impegna il risultato; `CMD` non esegue nulla al momento della compilazione, ma specifica il comando previsto per l'immagine.

Istruzione MAINTAINER

```
MAINTAINER <name>
```

L'istruzione `MAINTAINER` consente di impostare il campo Autore delle immagini generate.

NON USARE LA DIRETTIVA MAINTAINER

Secondo la documentazione ufficiale di Docker, l'istruzione `MAINTAINER` è deprecata. Invece, si dovrebbe usare l'istruzione `LABEL` per definire l'autore delle immagini generate. L'istruzione `LABEL` è più flessibile, consente di impostare i metadati e può essere facilmente visualizzata con il comando `docker inspect` .

```
LABEL maintainer="someone@something.com"
```

Dall'istruzione

```
FROM <image>
```

O

```
FROM <image>:<tag>
```

O

```
FROM <image>@<digest>
```

L'istruzione `FROM` imposta l'immagine di base per le istruzioni successive. Come tale, un Dockerfile valido deve avere `FROM` come prima istruzione. L'immagine può essere qualsiasi immagine valida - è particolarmente facile iniziare tirando un'immagine dai repository pubblici.

`FROM` deve essere la prima istruzione non commentata nel Dockerfile.

`FROM` può apparire più volte all'interno di un singolo Dockerfile per creare più immagini. Basta prendere nota dell'ultimo ID immagine emesso dal commit prima di ogni nuovo comando `FROM` .

I valori del tag o digest sono opzionali. Se si omette uno di essi, il builder assume un valore più recente per impostazione predefinita. Il builder restituisce un errore se non può corrispondere al valore del tag.

Istruzione RUN

`RUN` ha 2 forme:

```
RUN <command> (shell form, the command is run in a shell, which by default is /bin/sh -c on
Linux or cmd /S /C on Windows)
RUN ["executable", "param1", "param2"] (exec form)
```

L'istruzione `RUN` eseguirà tutti i comandi in un nuovo livello sopra l'immagine corrente e confermerà i risultati. L'immagine commessa risultante verrà utilizzata per il passaggio successivo nel file `Dockerfile` .

La stratificazione delle istruzioni `RUN` e la generazione dei commit sono conformi ai concetti chiave di Docker, in cui i commit sono convenienti e i contenitori possono essere creati da qualsiasi punto nella cronologia di un'immagine, proprio come il controllo del codice sorgente.

Il modulo exec consente di evitare il munging della stringa di shell e di eseguire comandi `RUN` utilizzando un'immagine di base che non contiene l'eseguibile shell specificato.

La shell di default per il modulo shell può essere modificata usando il comando `SHELL` .

Nel modulo della shell è possibile utilizzare un `\` (backslash) per continuare una singola istruzione `RUN` sulla riga successiva. Ad esempio, considera queste due linee:

```
RUN /bin/bash -c 'source $HOME/.bashrc ;\
echo $HOME'
```

Insieme sono equivalenti a questa singola riga:

```
RUN /bin/bash -c 'source $HOME/.bashrc ; echo $HOME'
```

Nota: per utilizzare una shell diversa, diversa da '/ bin / sh', utilizzare il modulo exec che passa nella shell desiderata. Ad esempio, `RUN ["/bin/bash", "-c", "echo hello"]`

Nota: il modulo exec viene analizzato come un array JSON, il che significa che è necessario utilizzare virgolette (`"`) attorno a words not single-quotes (`'`).

Nota: a differenza del modulo shell, il modulo exec non richiama una shell di comando. Ciò significa che la normale elaborazione della shell non avviene. Ad esempio, `RUN [ "echo", "$HOME" ]` non eseguirà la sostituzione delle variabili su `$HOME` . Se si desidera l'elaborazione della shell, utilizzare il modulo di shell o eseguire direttamente una shell, ad esempio: `RUN [ "sh", "-c", "echo $HOME" ]` .

Nota: nel modulo JSON, è necessario evitare i backslash. Questo è particolarmente rilevante su Windows in cui il backslash è il separatore del percorso. La seguente riga verrebbe altrimenti trattata come una forma di shell a causa di non essere JSON valido e non riuscire in modo inaspettato: `RUN ["c:\windows\system32\tasklist.exe"]`

La sintassi corretta per questo esempio è: `RUN ["c:\\windows\\system32\\tasklist.exe"]`

La cache per le istruzioni `RUN` non viene invalidata automaticamente durante la build successiva. La cache per un'istruzione come `RUN apt-get dist-upgrade -y` verrà riutilizzata durante la build successiva. La cache per le istruzioni `RUN` può essere invalidata utilizzando il flag --no-cache, ad esempio build docker --no-cache.

Per ulteriori informazioni, consultare la guida Best Practices di Dockerfile.

La cache per istruzioni `RUN` può essere invalidata dalle istruzioni `ADD` . Vedi sotto per i dettagli.

ONBUILD Istruzioni

```
ONBUILD [INSTRUCTION]
```

L'istruzione `ONBUILD` aggiunge all'immagine un'istruzione trigger da eseguire in un secondo momento, quando l'immagine viene utilizzata come base per un'altra build. Il trigger verrà eseguito nel contesto della build downstream, come se fosse stato inserito immediatamente dopo l'istruzione `FROM` nel Dockerfile downstream.

Qualsiasi istruzione di costruzione può essere registrata come trigger.

Ciò è utile se si sta costruendo un'immagine che verrà utilizzata come base per creare altre immagini, ad esempio un ambiente di sviluppo dell'applicazione o un demone che può essere personalizzato con una configurazione specifica dell'utente.

Ad esempio, se l'immagine è un costruttore di applicazioni Python riutilizzabile, richiederà l'aggiunta di un codice sorgente dell'applicazione in una directory specifica e potrebbe richiedere la creazione di uno script di generazione. Non puoi semplicemente chiamare `ADD` e `RUN` ora, perché non hai ancora accesso al codice sorgente dell'applicazione, e sarà diverso per ogni build dell'applicazione. Potresti semplicemente fornire agli sviluppatori di applicazioni un dockerfile di tipo standard per copiare e incollare nella loro applicazione, ma questo è inefficiente, soggetto a errori e difficile da aggiornare perché si mescola con il codice specifico dell'applicazione.

La soluzione è utilizzare `ONBUILD` per registrare le istruzioni avanzate da eseguire più tardi, durante la fase di costruzione successiva.

Ecco come funziona:

Quando incontra un'istruzione `ONBUILD` , il builder aggiunge un trigger ai metadati dell'immagine che viene creata. L'istruzione non influisce altrimenti sulla build corrente.

Alla fine della compilazione, un elenco di tutti i trigger è archiviato nel manifest dell'immagine, sotto la chiave OnBuild. Possono essere ispezionati con il comando di `docker inspect` . Successivamente l'immagine può essere usata come base per una nuova build, usando l'istruzione `FROM` . Come parte dell'elaborazione dell'istruzione `FROM` , il builder downstream cerca i trigger `ONBUILD` e li esegue nello stesso ordine in cui sono stati registrati. Se uno dei trigger fallisce, l'istruzione `FROM` viene interrotta, il che a sua volta causa il fallimento della build. Se tutti i trigger hanno esito positivo, l'istruzione `FROM` completata e la generazione continua come al solito.

I trigger vengono cancellati dall'immagine finale dopo essere stati eseguiti. In altre parole, non sono ereditati da build "grand-children".

Ad esempio potresti aggiungere qualcosa come questo:

```
[...]
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
[...]
```

Avviso: concatenare `ONBUILD` istruzioni `ONBUILD` utilizzando `ONBUILD ONBUILD` non è consentito.

Avvertenza: l'istruzione `ONBUILD` potrebbe non attivare le istruzioni `FROM` o `MAINTAINER` .

Istruzione STOPSIGNAL

```
STOPSIGNAL signal
```

L'istruzione `STOPSIGNAL` imposta il segnale di chiamata di sistema che verrà inviato al contenitore per uscire. Questo segnale può essere un numero non firmato valido che corrisponde a una posizione nella tabella syscall del kernel, ad esempio 9, o un nome di segnale nel formato SIGNAME, ad esempio SIGKILL.

Istruzione HEALTHCHECK

L'istruzione `HEALTHCHECK` ha due forme:

```
HEALTHCHECK [OPTIONS] CMD command (check container health by running a command inside the
container)
HEALTHCHECK NONE (disable any healthcheck inherited from the base image)
```

L'istruzione `HEALTHCHECK` dice a Docker come testare un contenitore per verificare che funzioni ancora. Questo può rilevare casi come un server Web bloccato in un ciclo infinito e incapace di gestire nuove connessioni, anche se il processo del server è ancora in esecuzione.

Quando un container ha un healthcheck specificato, ha uno stato di salute oltre al suo stato normale. Questo stato inizia inizialmente. Ogni volta che passa un controllo sanitario, diventa sano (qualunque stato fosse precedentemente). Dopo un certo numero di fallimenti consecutivi, diventa

malsano.

Le opzioni che possono apparire prima di `CMD` sono:

```
--interval=DURATION (default: 30s)
--timeout=DURATION (default: 30s)
--retries=N (default: 3)
```

Il controllo dello stato verrà eseguito per primi secondi dopo che il contenitore è stato avviato, e quindi di nuovo i secondi dopo il completamento di ogni controllo precedente.

Se una singola esecuzione del controllo richiede più tempo di un timeout, il controllo viene considerato non riuscito.

È necessario riprovare i fallimenti consecutivi del controllo dello stato perché il contenitore sia considerato non sano.

Ci può essere solo un'istruzione `HEALTHCHECK` in un `Dockerfile` . Se ne elenchiamo più di uno, solo l'ultimo `HEALTHCHECK` avrà effetto.

Il comando dopo la parola chiave `CMD` può essere un comando di shell (ad es. `HEALTHCHECK CMD /bin/check-running`) o un array exec (come con altri comandi Dockerfile, vedere ad esempio `ENTRYPOINT` per i dettagli).

Lo stato di uscita del comando indica lo stato di salute del contenitore. I valori possibili sono:

- `0: success` : il contenitore è integro e pronto all'uso
- `1: unhealthy` : il contenitore non funziona correttamente
- `2: starting` - il contenitore non è ancora pronto per l'uso, ma funziona correttamente

Se il probe restituisce 2 ("starting") quando il contenitore si è già spostato dallo stato "starting", viene invece considerato "malsano".

Ad esempio, per verificare ogni cinque minuti circa che un server Web sia in grado di servire la pagina principale del sito entro tre secondi:

```
HEALTHCHECK --interval=5m --timeout=3s \
  CMD curl -f http://localhost/ || exit 1
```

Per aiutare a eseguire il debug dei probe in errore, qualsiasi testo di output (codificato UTF-8) che il comando scrive su stdout o stderr verrà archiviato nello stato di `docker inspect` e può essere interrogato con il `docker inspect` . Tale output deve essere mantenuto breve (solo i primi 4096 byte sono attualmente memorizzati).

Quando cambia lo stato di salute di un contenitore, viene generato un evento `health_status` con il nuovo stato.

La funzione `HEALTHCHECK` è stata aggiunta a Docker 1.12.

SHELL Istruzione

```
SHELL ["executable", "parameters"]
```

L'istruzione `SHELL` consente di sovrascrivere la shell di default utilizzata per la forma shell dei comandi. La shell predefinita su Linux è `["/bin/sh", "-c"]` , e su Windows è `["cmd", "/S", "/C"]` . L'istruzione `SHELL` deve essere scritta in formato JSON in un Dockerfile.

L'istruzione `SHELL` è particolarmente utile su Windows in cui esistono due shell native comunemente usate e abbastanza diverse: cmd e powershell, oltre a shell alternative disponibili tra cui sh.

L'istruzione `SHELL` può apparire più volte. Ogni istruzione `SHELL` sostituisce tutte le precedenti istruzioni `SHELL` e influisce su tutte le istruzioni successive. Per esempio:

```
FROM windowsservercore

# Executed as cmd /S /C echo default
RUN echo default

# Executed as cmd /S /C powershell -command Write-Host default
RUN powershell -command Write-Host default

# Executed as powershell -command Write-Host hello
SHELL ["powershell", "-command"]
RUN Write-Host hello

# Executed as cmd /S /C echo hello
SHELL ["cmd", "/S"", "/C"]
RUN echo hello
```

Le seguenti istruzioni possono essere influenzate dall'istruzione `SHELL` quando la forma della shell di esse viene utilizzata in un Dockerfile: `RUN` , `CMD` e `ENTRYPOINT` .

L'esempio seguente è un modello comune trovato su Windows che può essere ottimizzato usando l'istruzione `SHELL` :

```
...
RUN powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"
...
```

Il comando invocato dalla finestra mobile sarà:

```
cmd /S /C powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"
```

Questo è inefficiente per due ragioni. Innanzitutto, viene invocato un comando del comando cmd.exe non necessario (noto anche come shell). In secondo luogo, ogni istruzione `RUN` nel modulo shell richiede un comando PowerShell supplementare che precede il comando.

Per rendere questo più efficiente, è possibile utilizzare uno dei due meccanismi. Uno è quello di utilizzare il modulo JSON del comando `RUN` come:

```
...
```

```
RUN ["powershell", "-command", "Execute-MyCmdlet", "-param1 \"c:\\foo.txt\""]
...
```

Mentre il modulo JSON non è ambiguo e non usa il cmd.exe non necessario, richiede più verbosità attraverso la doppia citazione e l'escaping. Il meccanismo alternativo consiste nell'utilizzare l'istruzione `SHELL` e la forma della shell, creando una sintassi più naturale per gli utenti di Windows, specialmente se combinata con la direttiva escape parser:

```
# escape=`

FROM windowsservercore
SHELL ["powershell","-command"]
RUN New-Item -ItemType Directory C:\Example
ADD Execute-MyCmdlet.ps1 c:\example\
RUN c:\example\Execute-MyCmdlet -sample 'hello world'
```

Con il risultato di:

```
PS E:\docker\build\shell> docker build -t shell .
Sending build context to Docker daemon 3.584 kB
Step 1 : FROM windowsservercore
 ---> 5bc36a335344
Step 2 : SHELL powershell -command
 ---> Running in 87d7a64c9751
 ---> 4327358436c1
Removing intermediate container 87d7a64c9751
Step 3 : RUN New-Item -ItemType Directory C:\Example
 ---> Running in 3e6ba16b8df9


Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         6/2/2016   2:59 PM                Example


 ---> 1f1dfdcec085
Removing intermediate container 3e6ba16b8df9
Step 4 : ADD Execute-MyCmdlet.ps1 c:\example\
 ---> 6770b4c17f29
Removing intermediate container b139e34291dc
Step 5 : RUN c:\example\Execute-MyCmdlet -sample 'hello world'
 ---> Running in abdcf50dfd1f
Hello from Execute-MyCmdlet.ps1 - passed hello world
 ---> ba0e25255fda
Removing intermediate container abdcf50dfd1f
Successfully built ba0e25255fda
PS E:\docker\build\shell>
```

L'istruzione `SHELL` potrebbe anche essere utilizzata per modificare il modo in cui una shell funziona. Ad esempio, utilizzando `SHELL cmd /S /C /V:ON|OFF` su Windows, è possibile modificare la semantica di espansione delle variabili di ambiente ritardate.

L'istruzione `SHELL` può anche essere utilizzata su Linux se è richiesta una shell alternativa come

zsh, csh, tcsh e altri.

La funzione `SHELL` è stata aggiunta a Docker 1.12.

Installazione dei pacchetti Debian / Ubuntu

Eseguire l'installazione su un comando a esecuzione singola per unire l'aggiornamento e l'installazione. Se aggiungi altri pacchetti in un secondo momento, questo eseguirà di nuovo l'aggiornamento e installerà tutti i pacchetti necessari. Se l'aggiornamento viene eseguito separatamente, verrà memorizzato nella cache e le installazioni del pacchetto potrebbero non riuscire. Impostare il frontend su non interattivo e passare il -y per installare è necessario per le installazioni con script. La pulizia e lo spurgo alla fine dell'installazione riducono al minimo le dimensioni del livello.

```
FROM debian

RUN apt-get update \
 && DEBIAN_FRONTEND=noninteractive apt-get install -y \
    git \
    openssh-client \
    sudo \
    vim \
    wget \
 && apt-get clean \
 && rm -rf /var/lib/apt/lists/*
```

Leggi Dockerfiles online: https://riptutorial.com/it/docker/topic/3161/dockerfiles

Capitolo 15: Esecuzione di contenitori

Sintassi

- finestra mobile [OPZIONI] IMMAGINE [COMANDO] [ARG ...]

Examples

Esecuzione di un contenitore

```
docker run hello-world
```

Questo recupererà l'ultima immagine ciao-mondo dall'Hub Docker (se non lo hai già), crea un nuovo contenitore ed eseguilo. Dovresti vedere un messaggio che informa che l'installazione sembra funzionare correttamente.

Esecuzione di un comando diverso nel contenitore

```
docker run docker/whalesay cowsay 'Hello, StackExchange!'
```

Questo comando dice a Docker di creare un contenitore dall'immagine `docker/whalesay` ed eseguire il comando `cowsay 'Hello, StackExchange!'` dentro. Dovrebbe stampare un'immagine di una balena che dice `Hello, StackExchange!` al tuo terminale.

Se il punto di inserimento nell'immagine è il valore predefinito, puoi eseguire qualsiasi comando disponibile nell'immagine:

```
docker run docker/whalesay ls /
```

Se è stato modificato durante la creazione dell'immagine, è necessario ripristinarlo di default

```
docker run --entrypoint=/bin/bash docker/whalesay -c ls /
```

Elimina automaticamente un contenitore dopo averlo eseguito

Normalmente, un contenitore Docker persiste dopo che è stato chiuso. Ciò consente di eseguire nuovamente il contenitore, ispezionare il suo filesystem e così via. Tuttavia, a volte si desidera eseguire un contenitore ed eliminarlo immediatamente dopo l'uscita. Ad esempio per eseguire un comando o mostrare un file dal filesystem. Docker fornisce l'opzione della riga di comando `--rm` per questo scopo:

```
docker run --rm ubuntu cat /etc/hosts
```

Questo creerà un contenitore dall'immagine "ubuntu", mostrerà il contenuto del **file / etc / hosts** e

quindi eliminerà il contenitore immediatamente dopo l'uscita. Questo aiuta a evitare di dover ripulire i contenitori dopo aver finito di sperimentare.

> **Nota:** il flag `--rm` non funziona insieme al flag `-d` (`--detach`) nella finestra mobile <1.13.0.

Quando viene impostato il flag `--rm` , Docker rimuove anche i volumi associati al contenitore quando il contenitore viene rimosso. È simile alla `docker rm -v my-container` in esecuzione. ***Vengono rimossi solo i volumi specificati senza un nome*** .

Ad esempio, con la `docker run -it --rm -v /etc -v logs:/var/log centos /bin/produce_some_logs` , il volume di `/etc` verrà rimosso, ma il volume di `/var/log` non lo sarà. I volumi ereditati tramite --volumes-from verranno rimossi con la stessa logica: se il volume originale è stato specificato con un nome, non verrà rimosso.

Specifica di un nome

Per impostazione predefinita, i contenitori creati con la `docker run` vengono assegnati con un nome casuale come `small_roentgen` o `modest_dubinsky` . Questi nomi non sono particolarmente utili per identificare lo scopo di un contenitore. È possibile fornire un nome per il contenitore passando l'opzione della riga di comando `--name` :

```
docker run --name my-ubuntu ubuntu:14.04
```

I nomi devono essere unici; se si passa un nome già utilizzato da un altro contenitore, Docker stamperà un errore e non verrà creato alcun nuovo contenitore.

Specificare un nome sarà utile quando si fa riferimento al contenitore all'interno di una rete Docker. Funziona sia per i contenitori Docker in background che in primo piano.

I contenitori sulla rete bridge predefinita **devono** essere collegati per comunicare per nome.

Associazione di una porta del contenitore all'host

```
docker run -p "8080:8080" myApp
docker run -p "192.168.1.12:80:80" nginx
docker run -P myApp
```

Per poter utilizzare le porte sull'host sono stati esposti in un'immagine (tramite il `EXPOSE` direttiva Dockerfile, o `--expose` linea di comando per `docker run`), le porte devono essere vincolato all'host utilizzando il `-p` o `-P` comando opzioni di linea. L'utilizzo di `-p` richiede che venga specificata la porta specifica (e l'interfaccia host facoltativa). L'uso dell'opzione della riga di comando maiuscola `-P` costringe Docker a collegare all'host *tutte* le porte esposte nell'immagine di un contenitore.

Politica di riavvio del contenitore (avvio di un contenitore all'avvio)

```
docker run --restart=always -d <container>
```

Per impostazione predefinita, Docker non riavvia i contenitori quando il daemon Docker viene riavviato, ad esempio dopo il riavvio del sistema host. Docker fornisce una politica di riavvio per i contenitori fornendo l'opzione della riga di comando `--restart` . Fornendo `--restart=always` causerà il riavvio di un container dopo il riavvio del daemon Docker. **Tuttavia,** quando quel container viene fermato manualmente (ad es. Con `docker stop <container>`), il criterio di riavvio non verrà applicato al contenitore.

È possibile specificare più opzioni per `--restart` opzione `--restart` , in base al requisito (`--restart=[policy]`). Queste opzioni influenzano il modo in cui il contenitore inizia anche all'avvio.

Politica	Risultato
no	Il valore **predefinito** Non riavvierà il contenitore automaticamente, quando il contenitore viene fermato.
on-fallimento [: max-tentativi]	Riavvia solo se il contenitore esce con un errore (`non-zero exit status`). Per evitare di riavviarlo indefinitamente (in caso di problemi), è possibile limitare il numero di tentativi di riavvio dei tentativi del daemon Docker.
sempre	Riavvia sempre il contenitore indipendentemente dallo stato di uscita. Quando si specifica `always` , il daemon Docker tenterà di riavviare il contenitore indefinitamente. Il contenitore verrà sempre avviato all'avvio del daemon, indipendentemente dallo stato corrente del contenitore.
a meno che non-fermato	Riavvia sempre il contenitore indipendentemente dal suo stato di uscita, ma non avviarlo all'avvio del daemon se il contenitore è stato precedentemente messo in stato di arresto.

Esegui un contenitore in background

Per mantenere un contenitore in esecuzione in background, fornire l'opzione della riga di comando `-d` durante l'avvio del contenitore:

```
docker run -d busybox top
```

L'opzione `-d` esegue il contenitore in modalità indipendente. È anche equivalente a `-d=true` .

Un contenitore in modalità distaccata non può essere rimosso automaticamente quando si ferma, questo significa che non si può usare l'opzione --rm in combinazione con l'opzione -d.

Assegna un volume a un contenitore

Un volume Docker è un file o una directory che persiste oltre la durata del contenitore. È possibile montare un file o una directory host in un contenitore come volume (ignorando UnionFS).

Aggiungi un volume con l'opzione della riga di comando `-v` :

```
docker run -d -v "/data" awesome/app bootstrap.sh
```

Questo creerà un volume e lo monterà sul percorso `/data` all'interno del contenitore.

- Nota: puoi usare il flag `--rm` per rimuovere automaticamente il volume quando il contenitore viene rimosso.

Montare le directory host

Per montare un file o una directory host in un contenitore:

```
docker run -d -v "/home/foo/data:/data" awesome/app bootstrap.sh
```

- **Quando si specifica una directory host, è necessario fornire un percorso assoluto.**

Questo monterà la directory host `/home/foo/data` su `/data` all'interno del contenitore. Questo volume "directory host bind-montato" è la stessa cosa di un `mount --bind` Linux `mount --bind` e quindi installa temporaneamente la directory host sul percorso contenitore specificato per la durata della vita del contenitore. Le modifiche nel volume dall'host o dal contenitore si riflettono immediatamente nell'altra, poiché sono la stessa destinazione su disco.

Esempio UNIX che monta una cartella relativa

```
docker run -d -v $(pwd)/data:/data awesome/app bootstrap.sh
```

Denominare i volumi

Un volume può essere denominato fornendo una stringa anziché un percorso di directory host, la finestra mobile crea un volume utilizzando tale nome.

```
docker run -d -v "my-volume:/data" awesome/app bootstrap.sh
```

Dopo aver creato un volume con nome, il volume può essere condiviso con altri contenitori usando quel nome.

Impostazione delle variabili di ambiente

```
$ docker run -e "ENV_VAR=foo" ubuntu /bin/bash
```

Sia `-e` che `--env` possono essere usati per definire le variabili d'ambiente all'interno di un contenitore. È possibile fornire molte variabili d'ambiente usando un file di testo:

```
$ docker run --env-file ./env.list ubuntu /bin/bash
```

Esempio di file delle variabili di ambiente:

```
# This is a comment
TEST_HOST=10.10.0.127
```

Il `--env-file` accetta un nome file come argomento e si aspetta che ogni riga sia nel formato `VARIABLE=VALUE` , imitando l'argomento passato a `--env` . Le righe di commento devono essere precedute solo da `#` .

Indipendentemente dall'ordine di questi tre flag, il `--env-file` viene elaborato per primo, quindi i flag `-e` / `--env` . In questo modo, qualsiasi variabile di ambiente fornita singolarmente con `-e` o `--env` sovrascriverà le variabili fornite nel file di testo `--env-var` .

Specifica di un nome host

Per impostazione predefinita, i contenitori creati con la finestra mobile vengono assegnati con un nome host casuale. Puoi dare al contenitore un nome host diverso passando il flag --hostname:

```
docker run --hostname redbox -d ubuntu:14.04
```

Esegui un contenitore in modo interattivo

Per eseguire un contenitore in modo interattivo, passa le opzioni `-it` :

```
$ docker run -it ubuntu:14.04 bash
root@8ef2356d919a:/# echo hi
hi
root@8ef2356d919a:/#
```

`-i` mantiene aperto STDIN, mentre `-t` alloca uno pseudo-TTY.

Contenitore in esecuzione con limiti di memoria / scambio

Imposta limite di memoria e disabilita il limite di swap

```
docker run -it -m 300M --memory-swap -1 ubuntu:14.04 /bin/bash
```

Imposta sia la memoria che il limite di swap. In questo caso, container può utilizzare 300M di memoria e 700M di swap.

```
docker run -it -m 300M --memory-swap 1G ubuntu:14.04 /bin/bash
```

Ottenere una shell in un contenitore (distaccato) in esecuzione

Accedere a un contenitore in esecuzione

Un utente può inserire un contenitore in esecuzione in una nuova shell bash interattiva con il comando **`exec`** .

Supponiamo che un contenitore sia chiamato `jovial_morse` quindi è possibile ottenere una bash shell pseudo-TTY interattiva eseguendo:

```
docker exec -it jovial_morse bash
```

Accedere a un contenitore in esecuzione con un utente specifico

Se si desidera immettere un contenitore come utente specifico, è possibile impostarlo con il parametro `-u` o `--user` . Il nome utente deve esistere nel contenitore.

> `-u, --user` Nome utente o UID (formato: `<name|uid>[:<group|gid>]`)

Questo comando si `jovial_morse` a `jovial_morse` con l'utente `dockeruser`

```
docker exec -it -u dockeruser jovial_morse bash
```

Accedere a un contenitore in esecuzione come root

Se vuoi accedere come root, usa semplicemente il parametro **`-u root`** . L'utente root esiste sempre.

```
docker exec -it -u root jovial_morse bash
```

Accedi ad un'immagine

Puoi anche accedere a un'immagine con il comando `run` , ma questo richiede un nome immagine invece di un nome contenitore.

```
docker run -it dockerimage bash
```

Accedi ad un'immagine intermedia (debug)

Puoi anche accedere a un'immagine intermedia, creata durante la creazione di un Dockerfile.

Output della `docker build .` della `docker build .`

```
$ docker build .
Uploading context 10240 bytes
Step 1 : FROM busybox
Pulling repository busybox
 ---> e9aa60c60128MB/2.284 MB (100%) endpoint: https://cdn-registry-1.docker.io/v1/
Step 2 : RUN ls -lh /
```

```
 ---> Running in 9c9e81692ae9
total 24
drwxr-xr-x    2 root     root        4.0K Mar 12  2013 bin
drwxr-xr-x    5 root     root        4.0K Oct 19 00:19 dev
drwxr-xr-x    2 root     root        4.0K Oct 19 00:19 etc
drwxr-xr-x    2 root     root        4.0K Nov 15 23:34 lib
lrwxrwxrwx    1 root     root           3 Mar 12  2013 lib64 -> lib
dr-xr-xr-x  116 root     root           0 Nov 15 23:34 proc
lrwxrwxrwx    1 root     root           3 Mar 12  2013 sbin -> bin
dr-xr-xr-x   13 root     root           0 Nov 15 23:34 sys
drwxr-xr-x    2 root     root        4.0K Mar 12  2013 tmp
drwxr-xr-x    2 root     root        4.0K Nov 15 23:34 usr
 ---> b35f4035db3f
Step 3 : CMD echo Hello world
 ---> Running in 02071fceb21b
 ---> f52f38b7823e
```

Si noti che `---> Running in 02071fceb21b` nell'output `---> Running in 02071fceb21b` , è possibile accedere a queste immagini:

```
docker run -it 02071fceb21b bash
```

Passando stdin al contenitore

In casi come il ripristino di un dump del database, o altrimenti che desideri spingere alcune informazioni attraverso una pipe dall'host, è possibile utilizzare il flag `-i` come argomento per `docker run` `docker exec` o `docker exec` .

Ad esempio, supponendo di voler mettere su un client mariadb containerizzato un dump del database che si ha sull'host, in un file `dump.sql` locale, è possibile eseguire il seguente comando:

```
docker exec -i mariadb bash -c 'mariadb "-p$MARIADB_PASSWORD" ' < dump.sql
```

In generale,

```
docker exec -i container command < file.stdin
```

O

```
docker exec -i container command <<EOF
inline-document-from-host-shell-HEREDOC-syntax
EOF
```

Scollegamento da un contenitore

Mentre è collegato a un contenitore in esecuzione con una pty assegnata (funzione `docker run -it ...`), è possibile premere `Control P` - `Control Q` per staccare.

Sovrascrittura della direttiva del punto di inserimento dell'immagine

```
docker run --name="test-app" --entrypoint="/bin/bash" example-app
```

Questo comando sovrascriverà la direttiva `ENTRYPOINT` di `example-app` quando viene creata l' `test-app` del contenitore. La direttiva `CMD` dell'immagine rimarrà invariata se non diversamente specificato:

```
docker run --name="test-app" --entrypoint="/bin/bash" example-app /app/test.sh
```

Nell'esempio precedente, sia l' `ENTRYPOINT` che il `CMD` dell'immagine sono stati sostituiti. Questo processo contenitore diventa `/bin/bash /app/test.sh` .

Aggiungi la voce host al contenitore

```
docker run --add-host="app-backend:10.15.1.24" awesome-app
```

Questo comando aggiunge una voce al `/etc/hosts` del contenitore, che segue il formato `--add-host <name>:<address>` . In questo esempio, il `app-backend` nome verrà risolto in `10.15.1.24` . Ciò è particolarmente utile per legare insieme componenti di applicazioni diverse tra loro.

Impedisci il blocco del contenitore quando non ci sono comandi in esecuzione

Un contenitore si fermerà se nessun comando è in esecuzione in primo piano. L'utilizzo dell'opzione `-t` impedisce al contenitore di fermarsi, anche quando viene rimosso con l'opzione `-d` .

```
docker run -t -d debian bash
```

Fermare un contenitore

```
docker stop mynginx
```

Inoltre, l'id contenitore può anche essere utilizzato per arrestare il contenitore anziché il suo nome.

Questo fermerà un contenitore funzionante inviando il segnale SIGTERM e quindi il segnale SIGKILL se necessario.

Inoltre, il comando kill può essere utilizzato per inviare immediatamente un SIGKILL o qualsiasi altro segnale specificato usando l'opzione `-s` .

```
docker kill mynginx
```

Segnale specificato:

```
docker kill -s SIGINT mynginx
```

L'arresto di un contenitore non lo elimina. Usa la `docker ps -a` per vedere il tuo contenitore fermato.

Esegui un altro comando su un contenitore in esecuzione

Quando richiesto, puoi dire a Docker di eseguire comandi aggiuntivi su un contenitore già in esecuzione usando il comando `exec` . È necessario l'ID del contenitore che è possibile ottenere con la `docker ps` .

```
docker exec 294fbc4c24b3 echo "Hello World"
```

È possibile allegare una shell interattiva se si utilizza l'opzione `-it` .

```
docker exec -it 294fbc4c24b3 bash
```

Esecuzione di app GUI in un contenitore Linux

Per impostazione predefinita, un contenitore Docker non sarà in grado di *eseguire* un'applicazione GUI.

Prima di ciò, il socket X11 deve essere prima inoltrato al contenitore, quindi può essere utilizzato direttamente. Anche la variabile d'ambiente *DISPLAY* deve essere inoltrata:

```
docker run -v /tmp/.X11-unix:/tmp/.X11-unix -e DISPLAY=unix$DISPLAY <image-name>
```

All'inizio questo non funzionerà, dal momento che non abbiamo impostato le autorizzazioni sull'host del server X:

```
cannot connect to X server unix:0
```

Il modo più veloce (ma non il più sicuro) è quello di consentire l'accesso direttamente con:

```
xhost +local:root
```

Dopo aver terminato con il contenitore, possiamo tornare allo stato originale con:

```
xhost -local:root
```

Un altro (più sicuro) modo è quello di preparare un Dockerfile che costruirà una nuova immagine che utilizzerà le credenziali dell'utente per accedere al server X:

```
FROM <iamge-name>
MAINTAINER <you>

# Arguments picked from the command line!
ARG user
ARG uid
ARG gid
```

```
#Add new user with our credentials
ENV USERNAME ${user}
RUN useradd -m $USERNAME && \
        echo "$USERNAME:$USERNAME" | chpasswd && \
        usermod --shell /bin/bash $USERNAME && \
        usermod  --uid ${uid} $USERNAME && \
        groupmod --gid ${gid} $USERNAME

USER ${user}

WORKDIR /home/${user}
```

Quando si richiama la `docker build` dalla riga di comando, dobbiamo passare le variabili *ARG* che appaiono nel file Docker:

```
docker build --build-arg user=$USER --build-arg uid=$(id -u) --build-arg gid=$(id -g) -t <new-image-with-X11-enabled-name> -f <Dockerfile-for-X11> .
```

Ora, prima di generare un nuovo contenitore, dobbiamo creare un file xauth con autorizzazione di accesso:

```
xauth nlist $DISPLAY | sed -e 's/^..../ffff/' | xauth -f /tmp/.docker.xauth nmerge -
```

Questo file deve essere montato nel contenitore durante la sua creazione / esecuzione:

```
docker run -e DISPLAY=unix$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix -v
/tmp/.docker.xauth:/tmp/.docker.xauth:rw -e XAUTHORITY=/tmp/.docker.xauth
```

Leggi Esecuzione di contenitori online: https://riptutorial.com/it/docker/topic/679/esecuzione-di-contenitori

Capitolo 16: Esecuzione di semplice applicazione Node.js

Examples

Esecuzione di un'applicazione Node.js di base in un contenitore

L'esempio che ho intenzione di discutere presuppone che tu abbia un'installazione Docker che funziona nel tuo sistema e una conoscenza di base su come lavorare con Node.js. Se sei a conoscenza di come devi lavorare con Docker, dovrebbe essere evidente che il framework Node.js non deve essere installato sul tuo sistema, piuttosto dovremmo usare l' `latest` versione dell'immagine del `node` disponibile da Docker. Quindi, se necessario, è possibile scaricare l'immagine in anticipo con il comando comando di `docker pull node` . (Il comando `pulls` automaticamente l'ultima versione dell'immagine del `node` dalla finestra mobile.)

1. Procedere alla creazione di una directory in cui risiedono tutti i file dell'applicazione in uso. Creare un file `package.json` in questa directory che descrive l'applicazione e le dipendenze. Il tuo file `package.json` dovrebbe assomigliare a questo:

```
{
  "name": "docker_web_app",
  "version": "1.0.0",
  "description": "Node.js on Docker",
  "author": "First Last <first.last@example.com>",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.13.3"
  }
}
```

2. Se abbiamo bisogno di lavorare con Node.js di solito creiamo un file `server` che definisce un'applicazione web. In questo caso utilizziamo il framework `Express.js` (versione `4.13.3` poi). Un file `server.js` base sarebbe simile a questo:

```
var express = require('express');
var PORT = 8080;
var app = express();
app.get('/', function (req, res) {
  res.send('Hello world\n');
});

app.listen(PORT);
console.log('Running on http://localhost:' + PORT);
```

3. Per chi ha familiarità con Docker, ti verrebbe in mente un file `Dockerfile` . Un `Dockerfile` è un file di testo che contiene tutti i comandi necessari per creare un'immagine personalizzata

che è fatta su misura per la tua applicazione.

Crea un file di testo vuoto chiamato `Dockerfile` nella directory corrente. Il metodo per crearne uno è semplice in Windows. In Linux, è possibile eseguire il `touch Dockerfile` nella directory contenente tutti i file necessari per la propria applicazione. Apri il Dockerfile con qualsiasi editor di testo e aggiungi le seguenti linee:

```
FROM node:latest
RUN mkdir -p /usr/src/my_first_app
WORKDIR /usr/src/my_first_app
COPY package.json /usr/src/my_first_app/
RUN npm install
COPY . /usr/src/my_first_app
EXPOSE 8080
```

- `FROM node:latest` istruisce il daemon Docker da quale immagine vogliamo costruire. In questo caso utilizziamo l' `latest` versione del `node` immagine Docker ufficiale disponibile da Docker Hub .
- All'interno di questa immagine procediamo a creare una directory di lavoro che contenga tutti i file richiesti e chiediamo al demone di impostare questa directory come directory di lavoro desiderata per la nostra applicazione. Per questo aggiungiamo

  ```
  RUN mkdir -p /usr/src/my_first_app
  WORKDIR /usr/src/my_first_app
  ```

- Procederemo quindi all'installazione delle dipendenze dell'applicazione spostando prima il file `package.json` (che specifica le informazioni sull'app incluse le dipendenze) nella `/usr/src/my_first_app` nell'immagine. Lo facciamo da

  ```
  COPY package.json /usr/src/my_first_app/
  RUN npm install
  ```

- Quindi `COPY . /usr/src/my_first_app` per aggiungere tutti i file dell'applicazione e il codice sorgente alla directory di lavoro nell'immagine.
- Quindi usiamo la direttiva `EXPOSE` per istruire il daemon a rendere visibile la porta `8080` del contenitore risultante (tramite una mappatura da contenitore a host) poiché l'applicazione si collega alla porta `8080` .
- Nell'ultimo passaggio, chiediamo al daemon di eseguire il comando `node server.js` all'interno dell'immagine eseguendo il comando di base `npm start` . Usiamo la direttiva `CMD` per questo, che accetta i comandi come argomenti.

  ```
  CMD [ "npm", "start" ]
  ```

4. Creiamo quindi un file `.dockerignore` nella stessa directory del `Dockerfile` per impedire che la nostra copia di `node_modules` e log utilizzati dall'installazione del nostro sistema Node.js vengano copiati sull'immagine Docker. Il file `.dockerignore` deve avere il seguente contenuto:

```
node_modules
npm-debug.log
```

5. # Costruisci la tua immagine

Passare alla directory che contiene il `Dockerfile` ed eseguire il seguente comando per creare l'immagine Docker. Il flag `-t` ti consente di taggare la tua immagine in modo che sia più facile da trovare in seguito utilizzando il comando immagini docker:

```
    $ docker build -t <your username>/node-web-app .
```

L'immagine verrà ora elencata da Docker. Guarda le immagini usando il comando seguente:

```
$ docker images

REPOSITORY                         TAG         ID              CREATED
node                               latest      539c0211cd76    10 minutes ago
<your username>/node-web-app       latest      d64d3505b0d2    1 minute ago
```

6. # Esecuzione dell'immagine

Ora possiamo eseguire l'immagine appena creata utilizzando i contenuti dell'applicazione, l'immagine base del `node` e il `Dockerfile` . Ora procediamo con la nostra nuova immagine `<your username>/node-web-app` . Fornendo l' `-d` al comando di `docker run` della `docker run` , il contenitore viene eseguito in modalità indipendente, in modo che il contenitore `docker run` eseguito in background. Il flag `-p` reindirizza una porta pubblica a una porta privata all'interno del contenitore. Esegui l'immagine creata in precedenza utilizzando questo comando:

```
$ docker run -p 49160:8080 -d <your username>/node-web-app
```

7. Stampa l'output della tua app eseguendo la `docker ps` sul tuo terminale. L'output dovrebbe essere simile a questo.

```
 CONTAINER ID        IMAGE                          COMMAND             CREATED
STATUS              PORTS                      NAMES
 7b701693b294      <your username>/node-web-app   "npm start"         20 minutes ago
Up 48 seconds       0.0.0.0:49160->8080/tcp   loving_goldstine
```

Ottieni l'output dell'applicazione inserendo i `docker logs <CONTAINER ID>` . In questo caso si tratta dei `docker logs 7b701693b294` .

Uscita: in `Running on http://localhost:8080`

8. `docker ps` , la mappatura delle porte ottenuta è `0.0.0.0:49160->8080/tcp` . Quindi Docker ha mappato la porta `8080` all'interno del contenitore alla porta 49160 sul computer host. Nel browser possiamo ora inserire `localhost:49160` .

Possiamo anche chiamare la nostra app usando `curl` :

```
$ curl -i localhost:49160

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 12
Date: Sun, 08 Jan 2017 14:00:12 GMT
Connection: keep-alive

Hello world
```

Leggi Esecuzione di semplice applicazione Node.js online:
https://riptutorial.com/it/docker/topic/8754/esecuzione-di-semplice-applicazione-node-js

Capitolo 17: eseguire console in docker 1.12 sciame

Examples

Esegui console in un docker 1.12 sciame

Questo si basa sull'immagine ufficiale docker console per eseguire console in modalità cluster in uno sciame docker con nuova modalità sciame in Docker 1.12. Questo esempio è basato su http://qnib.org/2016/08/11/consul-service/ . In breve, l'idea è di utilizzare due servizi swarm docker che parlano tra loro. Questo risolve il problema che non puoi conoscere gli IP dei singoli container console in anticipo e ti consente di fare affidamento sui dns degli sciami docker.

Questo presuppone che tu abbia già un cluster sciame di 1.12 docker in esecuzione con almeno tre nodi.

Potrebbe essere necessario configurare un driver di registro sui daemon della finestra mobile in modo da poter vedere cosa sta succedendo. Ho usato il driver syslog per questo: imposta l' `--log-driver=syslog` su dockerd.

Per prima cosa crea una rete overlay per console:

```
docker network create consul-net -d overlay
```

Ora avvia il cluster con un solo nodo (default --replicas è 1):

```
docker service create --name consul-seed \
  -p 8301:8300 \
  --network consul-net \
  -e 'CONSUL_BIND_INTERFACE=eth0' \
  consul agent -server -bootstrap-expect=3  -retry-join=consul-seed:8301 -retry-join=consul-
cluster:8300
```

Ora dovresti avere un cluster a 1 nodo. Ora fai apparire il secondo servizio:

```
docker service create --name consul-cluster \
  -p 8300:8300 \
  --network consul-net \
  --replicas 3 \
  -e 'CONSUL_BIND_INTERFACE=eth0' \
  consul agent -server -retry-join=consul-seed:8301 -retry-join=consul-cluster:8300
```

Ora dovresti avere un console console a quattro nodi. Puoi verificarlo eseguendo su uno qualsiasi dei contenitori docker:

```
docker exec <containerid> consul members
```

Leggi eseguire console in docker 1.12 sciame online:
https://riptutorial.com/it/docker/topic/6437/eseguire-console-in-docker-1-12-sciame

Capitolo 18: Eventi Docker

Examples

Avvia un container e ricevi una notifica degli eventi correlati

La documentazione per gli `docker events` fornisce dettagli, ma quando si esegue il debug può essere utile avviare un contenitore e ricevere immediatamente la notifica di qualsiasi evento correlato:

```
docker run... & docker events --filter 'container=$(docker ps -lq)'
```

Nella `docker ps -lq` , la `l` sta per `last` , e la `q` per `quiet` . Ciò rimuove l' `id` dell'ultimo contenitore avviato e crea immediatamente una notifica se il contenitore muore o si verifica un altro evento.

Leggi Eventi Docker online: https://riptutorial.com/it/docker/topic/6200/eventi-docker

Capitolo 19: finestra mobile ispeziona i vari campi per la chiave: valore ed elementi della lista

Examples

vari docker ispezionano esempi

Trovo che gli esempi nel `docker inspect` documentazione sembrano magici, ma non spiegano molto.

L'ispezione di Docker è importante perché è il modo pulito per estrarre informazioni da una `docker inspect -f ... container_id` container in esecuzione `docker inspect -f ... container_id`

(o tutto il contenitore funzionante)

```
docker inspect -f ... $(docker ps -q)
```

evitando alcuni inaffidabili

```
docker command | grep or awk | tr or cut
```

Quando si avvia un `docker inspect` è possibile ottenere facilmente i valori dal "livello principale", con una sintassi di base come, per un contenitore che esegue htop (da https://hub.docker.com/r/jess/htop/) con un pid ae1

```
docker inspect -f '{{.Created}}' ae1
```

può mostrare

```
2016-07-14T17:44:14.159094456Z
```

o

```
docker inspect -f '{{.Path}}' ae1
```

può mostrare

```
htop
```

Ora se estraggo una parte del mio `docker inspect`

Vedo

```
"State": { "Status": "running", "Running": true, "Paused": false, "Restarting": false, "OOMKilled": false, "Dead": false, "Pid": 4525, "ExitCode": 0, "Error": "", "StartedAt": "2016-07-14T17:44:14.406286293Z", "FinishedAt": "0001-01-01T00:00:00Z"
```

Quindi ottengo un dizionario, come ha `{ ...}` e un sacco di chiavi: i valori

Quindi il comando

```
docker inspect -f '{{.State}}' ae1
```

restituirà una lista, come

```
{running true false false false false 4525 0 2016-07-14T17:44:14.406286293Z 0001-01-
01T00:00:00Z}
```

Posso ottenere facilmente il valore di State.Pid

```
docker inspect -f '{{ .State.Pid }}' ae1
```

ottengo

```
4525
```

A volte l'ispezione di una finestra mobile fornisce una lista cominciando da `[` e finisce con `]`

un altro esempio, con un altro contenitore

```
docker inspect -f '{{ .Config.Env }}' 7a7
```

dà

```
[DISPLAY=:0 PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin LANG=fr_FR.UTF-8
LANGUAGE=fr_FR:en LC_ALL=fr_FR.UTF-8 DEBIAN_FRONTEND=noninteractive HOME=/home/gg WINEARCH=win32
WINEPREFIX=/home/gg/.wine_captvty]
```

Per ottenere il primo elemento della lista, aggiungiamo indice prima del campo richiesto e 0 (come primo elemento) dopo, quindi

```
docker inspect -f '{{ index ( .Config.Env) 0 }}' 7a7
```

dà

```
DISPLAY=:0
```

Otteniamo il prossimo elemento con 1 invece di 0 usando la stessa sintassi

```
docker inspect -f '{{ index ( .Config.Env) 1 }}' 7a7
```

dà

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

Possiamo ottenere il numero di elementi di questa lista

```
docker inspect -f '{{ len .Config.Env }}' 7a7
```

dà

```
9
```

e possiamo ottenere l'ultimo elemento della lista, la sintassi non è facile

```
docker inspect -f "{{ index .Config.Cmd $[$(docker inspect -format '{{ len .Config.Cmd }}'
$CID)-1]}}" 7a7
```

Leggi finestra mobile ispeziona i vari campi per la chiave: valore ed elementi della lista online: https://riptutorial.com/it/docker/topic/6470/finestra-mobile-ispeziona-i-vari-campi-per-la-chiave--valore-ed-elementi-della-lista

Capitolo 20: Gestione dei contenitori

Sintassi

- docker rm [OPZIONI] CONTENITORE [CONTENITORE ...]
- docker allegare [OPZIONI] CONTENITORE
- docker exec [OPZIONI] CONTAINER COMMAND [ARG ...]
- docker ps [OPZIONI]
- log del docker [OPZIONI] CONTENITORE
- finestra mobile ispeziona [OPZIONI] CONTENITORE | IMMAGINE [CONTENITORE | IMMAGINE ...]

Osservazioni

- Negli esempi precedenti, ogni volta che container è un parametro del comando docker, viene indicato come `<container>` o `container id` o `<CONTAINER_NAME>` . In tutti questi posti è possibile passare un nome contenitore o un id contenitore per specificare un contenitore.

Examples

Elenco dei contenitori

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                     NAMES
2bc9b1988080        redis               "docker-entrypoint.sh"   2 weeks ago         Up 2
hours          0.0.0.0:6379->6379/tcp    elephant-redis
817879be2230        postgres            "/docker-entrypoint.s"   2 weeks ago         Up 2
hours          0.0.0.0:65432->5432/tcp   pt-postgres
```

`docker ps` da sola stampa solo i contenitori attualmente in esecuzione. Per visualizzare tutti i contenitori (compresi quelli fermati), usa il flag `-a` :

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                     NAMES
9cc69f11a0f7        docker/whalesay     "ls /"                   26 hours ago        Exited
(0) 26 hours ago                            berserk_wozniak
2bc9b1988080        redis               "docker-entrypoint.sh"   2 weeks ago         Up 2
hours                0.0.0.0:6379->6379/tcp    elephant-redis
817879be2230        postgres            "/docker-entrypoint.s"   2 weeks ago         Up 2
hours                0.0.0.0:65432->5432/tcp   pt-postgres
```

Per elencare i contenitori con uno stato specifico, utilizzare l'opzione della riga di comando `-f` per filtrare i risultati. Ecco un esempio di elenco di tutti i contenitori che sono usciti:

```
$ docker ps -a -f status=exited
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                     NAMES
9cc69f11a0f7        docker/whalesay     "ls /"                   26 hours ago        Exited
(0) 26 hours ago
```

È anche possibile elencare solo gli ID contenitore con l' `-q` . Questo rende molto facile operare sul risultato con altre utility Unix (come `grep` e `awk`):

```
$ docker ps -aq
9cc69f11a0f7
2bc9b1988080
817879be2230
```

Quando si avvia un container con la `docker run --name mycontainer1` si specifica un nome specifico e non un nome casuale (nella forma mood_famous, come nostalgic_stallman), e può essere facile trovarli con tale comando

```
docker ps -f name=mycontainer1
```

Contenitori di riferimento

I comandi Docker che prendono il nome di un contenitore accettano tre diverse forme:

genere	Esempio
UUID completo	`9cc69f11a0f76073e87f25cb6eaf0e079fbfbd1bc47c063bcd25ed3722a8cc4a`
UUID breve	`9cc69f11a0f7`
Nome	`berserk_wozniak`

Utilizzare la `docker ps` per visualizzare questi valori per i contenitori sul proprio sistema.

L'UUID è generato da Docker e non può essere modificato. È possibile fornire un nome al contenitore quando lo si avvia `docker run --name <given name> <image>` . Docker genererà un nome casuale nel contenitore se non ne specifichi uno al momento dell'avvio del contenitore.

***NOTA** : il valore dell'UUID (o un UUID "breve") può essere di qualsiasi lunghezza purché il valore specificato sia univoco per un contenitore*

Avvio e arresto dei contenitori

Per fermare un container in esecuzione:

```
docker stop <container> [<container>...]
```

Questo invierà il processo principale nel contenitore a SIGTERM, seguito da un SIGKILL se non si interrompe entro il periodo di prova. Il nome di ciascun contenitore viene stampato mentre si arresta.

Per avviare un contenitore interrotto:

```
docker start <container> [<container>...]
```

Questo avvierà ogni contenitore passato in background; il nome di ciascun contenitore viene stampato non appena inizia. Per avviare il contenitore in primo piano, passare il flag `-a` (`--attach`).

Elenca contenitori con formato personalizzato

```
docker ps --format 'table {{.ID}}\t{{.Names}}\t{{.Status}}'
```

Trovare un contenitore specifico

```
docker ps --filter name=myapp_1
```

Trova IP del contenitore

Per scoprire l'indirizzo IP del tuo contenitore, usa:

```
docker inspect <container id> | grep IPAddress
```

oppure utilizzare la finestra mobile ispeziona

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' ${CID}
```

Riavvio del contenitore della finestra mobile

```
docker restart <container> [<container>...]
```

Opzione - **tempo** : secondi per attendere l'arresto prima di uccidere il contenitore (impostazione predefinita 10)

```
docker restart <container> --time 10
```

Rimuovere, eliminare e pulire i contenitori

`docker rm` può essere usato per rimuovere contenitori specifici come questo:

```
docker rm <container name or id>
```

Per rimuovere tutti i contenitori puoi usare questa espressione:

```
docker rm $(docker ps -qa)
```

Per impostazione predefinita la finestra mobile non elimina un contenitore in esecuzione. Qualsiasi

contenitore in esecuzione produrrà un messaggio di avviso e non verrà eliminato. Tutti gli altri contenitori saranno cancellati.

In alternativa puoi usare `xargs` :

```
docker ps -aq -f status=exited | xargs -r docker rm
```

Dove la `docker ps -aq -f status=exited` restituirà un elenco di ID contenitore di container con stato "Exited".

> Attenzione: tutti gli esempi precedenti rimuoveranno solo i contenitori "fermati".

Per rimuovere un container, indipendentemente dal fatto che sia stato arrestato o meno, puoi utilizzare il flag force `-f` :

```
docker rm -f <container name or id>
```

Per rimuovere tutti i contenitori, indipendentemente dallo stato:

```
docker rm -f $(docker ps -qa)
```

Se si desidera rimuovere solo i contenitori con stato `dead` :

```
docker rm $(docker ps --all -q -f status=dead)
```

Se si desidera rimuovere solo i contenitori con uno stato `exited` :

```
docker rm $(docker ps --all -q -f status=exited)
```

Queste sono tutte permutazioni dei filtri utilizzati quando si elencano i contenitori .

Per rimuovere sia i contenitori indesiderati che le immagini che pendono dallo spazio dopo la versione 1.3 , utilizzare quanto segue (simile allo strumento Unix `df`):

```
$ docker system df
```

Per rimuovere tutti i dati inutilizzati:

```
$ docker system prune
```

Esegui il comando su un contenitore finestra mobile già esistente

```
docker exec -it <container id> /bin/bash
```

È normale accedere a un contenitore già in esecuzione per effettuare alcuni test rapidi o vedere cosa sta facendo l'applicazione. Spesso denota cattive pratiche di utilizzo del contenitore dovute ai log e i file modificati dovrebbero essere collocati in volumi. Questo esempio ci consente di accedere al contenitore. Ciò suppone che / bin / bash sia disponibile nel contenitore, che possa

essere / bin / sh o qualcos'altro.

```
docker exec <container id> tar -czvf /tmp/backup.tgz /data
docker cp <container id>:/tmp/backup.tgz .
```

Questo esempio archivia il contenuto della directory dei dati in un tar. Quindi con `docker cp` puoi recuperarlo.

Log del contenitore

```
Usage:  docker logs [OPTIONS] CONTAINER

Fetch the logs of a container

  -f, --follow=false        Follow log output
  --help=false              Print usage
  --since=                  Show logs since timestamp
  -t, --timestamps=false    Show timestamps
  --tail=all                Number of lines to show from the end of the logs
```

Per esempio:

```
$ docker ps
CONTAINER ID    IMAGE    COMMAND                  CREATED     STATUS       PORTS
ff9716dda6cb    nginx    "nginx -g 'daemon off"  8 days ago  Up 22 hours  443/tcp,
0.0.0.0:8080->80/tcp

$ docker logs ff9716dda6cb
xx.xx.xx.xx - - [15/Jul/2016:14:03:44 +0000] "GET /index.html HTTP/1.1" 200 511
"https://google.com" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/50.0.2661.75 Safari/537.36"
xx.xx.xx.xx - - [15/Jul/2016:14:03:44 +0000] "GET /index.html HTTP/1.1" 200 511
"https://google.com" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/50.0.2661.75 Safari/537.36"
```

Connettersi a un'istanza in esecuzione come daemon

Ci sono due modi per farlo, il primo e il più conosciuto è il seguente:

```
docker attach --sig-proxy=false <container>
```

Questo letteralmente attacca il tuo bash alla bash del contenitore, il che significa che se hai uno script in esecuzione, vedrai il risultato.

Per staccare, basta digitare: `Ctl-P Ctl-Q`

Ma se hai bisogno di un modo più amichevole e di essere in grado di creare nuove istanze di bash, basta eseguire il seguente comando:

```
docker exec -it <container> bash
```

Copia di file da / a contenitori

dal contenitore all'host

```
docker cp CONTAINER_NAME:PATH_IN_CONTAINER PATH_IN_HOST
```

dall'host al contenitore

```
docker cp PATH_IN_HOST CONTAINER_NAME:PATH_IN_CONTAINER
```

Se uso jess / trasmissione da

https://hub.docker.com/r/jess/transmission/builds/bsn7eqxrkzrhxazcuytbmzp/

, i file nel contenitore sono in / transmission / download

e la mia directory corrente sull'host è / home / $ USER / abc, dopo

```
docker cp transmission_id_or_name:/transmission/download .
```

Farò copiare i file in

```
/home/$USER/abc/transmission/download
```

non è possibile, utilizzando la `docker cp` copiare solo un file, copiare l'albero della directory e i file

Rimuovere, eliminare e pulire i volumi della finestra mobile

I volumi Docker non vengono rimossi automaticamente quando un container viene fermato. Per rimuovere i volumi associati quando si interrompe un contenitore:

```
docker rm -v <container id or name>
```

Se non viene specificato il flag `-v` , il volume rimane su disco come "volume sospeso". Per eliminare tutti i volumi che pendono:

```
docker volume rm $(docker volume ls -qf dangling=true)
```

Il `docker volume ls -qf dangling=true` filter restituirà un elenco di nomi di volumi di finestra mobile, inclusi quelli senza tag, che non sono collegati a un contenitore.

In alternativa, puoi usare `xargs` :

```
docker volume ls -f dangling=true -q | xargs --no-run-if-empty docker volume rm
```

Esportare e importare i filesystem del contenitore Docker

È possibile salvare il contenuto del file system di un contenitore Docker in un file di archivio tarball. Ciò è utile in un pizzico per spostare i filesystem del contenitore su diversi host, ad esempio se un

contenitore di database ha modifiche importanti e non è altrimenti possibile replicare tali modifiche altrove. **Si noti** che è preferibile creare un contenitore completamente nuovo da un'immagine aggiornata utilizzando un comando di `docker run docker-compose.yml` o `docker-compose.yml` file `docker-compose.yml` , invece di esportare e spostare il filesystem di un contenitore. Parte del potere di Docker è l'auditabilità e la responsabilità del suo stile dichiarativo di creazione di immagini e contenitori. Utilizzando `docker export` e `docker import` , questo potere è sottomesso a causa della offuscamento delle modifiche apportate all'interno del filesystem di un contenitore dal suo stato originale.

```
docker export -o redis.tar redis
```

Il comando precedente creerà un'immagine vuota e quindi esporterà il filesystem del contenitore `redis` in questa immagine vuota. Per importare da un archivio tarball, usa:

```
docker import ./redis.tar redis-imported:3.0.7
```

Questo comando creerà l'immagine `redis-imported:3.0.7` , da cui è possibile creare contenitori. È anche possibile creare modifiche all'importazione e impostare un messaggio di commit:

```
docker import -c="ENV DEBUG true" -m="enable debug mode" ./redis.tar redis-changed
```

Le direttive Dockerfile disponibili per l'uso con l' `-c` opzione della riga di comando sono `CMD` , `ENTRYPOINT` , `ENV` , `EXPOSE` , `ONBUILD` , `USER` , `VOLUME` , `WORKDIR` .

Leggi Gestione dei contenitori online: https://riptutorial.com/it/docker/topic/689/gestione-dei-contenitori

Capitolo 21: Gestire le immagini

Sintassi

- immagini docker [OPZIONI] [REPOSITORY [: TAG]]
- finestra mobile ispeziona [OPZIONI] CONTENITORE | IMMAGINE [CONTENITORE | IMMAGINE ...]
- docker pull [OPZIONI] NOME [: TAG | @DIGEST]
- docker rmi [OPZIONI] IMMAGINE [IMMAGINE ...]
- tag docker [OPZIONI] IMAGE [: TAG] [REGISTRYHOST /] [USERNAME /] NAME [: TAG]

Examples

Recupero di un'immagine dall'hub Docker

Normalmente, le immagini vengono estratte automaticamente da Docker Hub . Docker tenterà di estrarre qualsiasi immagine dall'hub Docker che non esiste già sull'host Docker. Ad esempio, utilizzando la `docker run ubuntu` quando l'immagine `ubuntu` non è già presente sull'host Docker, Docker avvierà un pull dell'ultima immagine `ubuntu` . È possibile estrarre un'immagine separatamente usando la funzione di `docker pull` per recuperare manualmente o aggiornare un'immagine da Docker Hub.

```
docker pull ubuntu
docker pull ubuntu:14.04
```

Esistono ulteriori opzioni per l'estrazione da un registro di immagine diverso o il prelievo di una versione specifica di un'immagine. L'indicazione di un registro alternativo viene eseguita utilizzando il nome completo dell'immagine e la versione opzionale. Ad esempio, il seguente comando tenterà di estrarre l'immagine `ubuntu:14.04` dal `registry.example.com` registry.example.com:

```
docker pull registry.example.com/username/ubuntu:14.04
```

Elenco delle immagini scaricate localmente

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello-world         latest              693bce725149        6 days ago          967 B
postgres            9.5                 0f3af79d8673        10 weeks ago        265.7 MB
postgres            latest              0f3af79d8673        10 weeks ago        265.7 MB
```

Fare riferimento alle immagini

I comandi Docker che prendono il nome di un'immagine accettano quattro diverse forme:

genere	Esempio
ID breve	`693bce725149`
Nome	`hello-world` *(predefinito su `:latest` tag)*
Nome + tag	`hello-world:latest`
digerire	`hello-world@sha256:e52be8ffeeb1f374f440893189cd32f44cb166650e7ab185fa7735b7dc48d619`

Nota: è possibile fare riferimento a un'immagine solo dal suo sommario se quell'immagine è stata originariamente tirata usando quel digest. Per visualizzare il digest per un'immagine (se disponibile), esegui `docker images --digests` .

Rimozione di immagini

Il comando `docker rmi` viene utilizzato per rimuovere le immagini:

```
docker rmi <image name>
```

Il nome completo dell'immagine deve essere usato per rimuovere un'immagine. A meno che l'immagine non sia stata taggata per rimuovere il nome del registro, è necessario specificarla. Per esempio:

```
docker rmi registry.example.com/username/myAppImage:1.3.5
```

È anche possibile rimuovere le immagini con il loro ID invece:

```
docker rmi 693bce725149
```

Per comodità, è possibile rimuovere le immagini con il loro ID immagine specificando solo i primi caratteri dell'ID immagine, a condizione che la sottostringa specificata non sia ambigua:

```
docker rmi 693
```

> **Nota: le** immagini possono essere rimosse anche se esistono contenitori esistenti che utilizzano quell'immagine; docker rmi semplicemente "disegna" l'immagine.

Se nessun contenitore utilizza un'immagine, viene raccolto dai rifiuti. Se un contenitore utilizza un'immagine, l'immagine verrà raccolta dopo che tutti i contenitori che lo utilizzano vengono rimossi. Per esempio:

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
PORTS               NAMES
5483657ee07b        hello-world         "/hello"            Less than a second ago   Exited
(0) 2 seconds ago                       small_elion
```

```
$ docker rmi hello-world
Untagged: hello-world:latest

$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
PORTS               NAMES
5483657ee07b        693bce725149        "/hello"            Less than a second ago   Exited
(0) 12 seconds ago                       small_elion
```

Rimuovi tutte le immagini senza contenitori avviati

Per rimuovere tutte le immagini locali senza contenitori avviati, puoi fornire un elenco delle immagini come parametro:

```
docker rmi $(docker images -qa)
```

Rimuovi tutte le immagini

Se si desidera rimuovere le immagini indipendentemente dal fatto che abbiano o meno un contenitore avviato, utilizzare il flag force (`-f`):

```
docker rmi -f $(docker images -qa)
```

Rimuovi le immagini penzolanti

Se un'immagine non è contrassegnata e non viene utilizzata da alcun contenitore, è "penzolante" e può essere rimossa in questo modo:

```
docker images -q --no-trunc -f dangling=true | xargs -r docker rmi
```

Cerca nell'hub Docker le immagini

Puoi cercare Docker Hub per immagini usando il comando di ricerca :

```
docker search <term>
```

Per esempio:

```
$ docker search nginx
NAME                     DESCRIPTION                                     STARS     OFFICIAL
AUTOMATED
nginx                    Official build of Nginx.                        3565      [OK]
jwilder/nginx-proxy      Automated Nginx reverse proxy for docker c...   717
[OK]
richarvey/nginx-php-fpm  Container running Nginx + PHP-FPM capable ...   232
[OK]
...
```

Ispezionando le immagini

```
docker inspect <image>
```

L'output è in formato JSON. È possibile utilizzare l'utilità della riga di comando `jq` per analizzare e stampare solo i tasti desiderati.

```
docker inspect <image> | jq -r '.[0].Author'
```

Il comando sopra mostrerà il nome dell'autore delle immagini.

Tagging delle immagini

Il tagging di un'immagine è utile per tenere traccia delle diverse versioni dell'immagine:

```
docker tag ubuntu:latest registry.example.com/username/ubuntu:latest
```

Un altro esempio di tagging:

```
docker tag myApp:1.4.2 myApp:latest
docker tag myApp:1.4.2 registry.example.com/company/myApp:1.4.2
```

Salvataggio e caricamento di immagini Docker

```
docker save -o ubuntu.latest.tar ubuntu:latest
```

Questo comando salverà l' `ubuntu:latest` immagine come archivio di tarball nella directory corrente con il nome `ubuntu.latest.tar` . Questo archivio tarball può quindi essere spostato su un altro host, ad esempio utilizzando `rsync` o archiviato in archivio.

Una volta che il tarball è stato spostato, il seguente comando creerà un'immagine dal file:

```
docker load -i /tmp/ubuntu.latest.tar
```

Ora è possibile creare contenitori da `ubuntu:latest` immagine come al solito.

Leggi Gestire le immagini online: https://riptutorial.com/it/docker/topic/690/gestire-le-immagini

Capitolo 22: Iptables con Docker

introduzione

Questo argomento riguarda come limitare l'accesso ai contenitori docker dal mondo esterno usando iptables.

Per le persone impazienti, puoi controllare gli esempi. Per gli altri, leggi la sezione delle note per capire come creare nuove regole.

Sintassi

- iptables -I DOCKER [RULE ...] [ACCEPT | DROP] // Per aggiungere una regola alla parte superiore della tabella DOCKER
- iptables -D DOCKER [RULE ...] [ACCEPT | DROP] // Per rimuovere una regola dalla tabella DOCKER
- ripristino ipset </etc/ipfriends.conf // Per riconfigurare i propri IPs *ipset*

Parametri

parametri	Dettagli
ext_if	La tua interfaccia esterna sull'host Docker.
XXX.XXX.XXX.XXX	Un particolare IP su cui deve essere fornito l'accesso ai contenitori Docker.
yyy.yyy.yyy.yyy	Un altro IP su cui devono essere forniti i contenitori Docker.
ipfriends	Il nome ipset che definisce gli IP autorizzati ad accedere ai contenitori Docker.

Osservazioni

Il problema

La configurazione delle regole di iptables per i contenitori Docker è un po 'complicata. All'inizio, penseresti che le regole del firewall "classiche" dovrebbero fare il trucco.

Ad esempio, supponiamo di aver configurato un contenitore proxy nginx + diversi contenitori di servizi per esporre via HTTPS alcuni servizi web personali. Quindi una regola come questa dovrebbe consentire l'accesso ai tuoi servizi Web solo per IP XXX.XXX.XXX.XXX.

```
$ iptables -A INPUT -i eth0 -p tcp -s XXX.XXX.XXX.XXX -j ACCEPT
$ iptables -P INPUT DROP
```

Non funzionerà, i tuoi contenitori sono ancora accessibili a tutti.

Infatti, i container Docker non sono servizi host. Si basano su una rete virtuale nel tuo host e l'host funge da gateway per questa rete. E per quanto riguarda i gateway, il traffico indirizzato non viene gestito dalla tabella INPUT, ma dalla tabella FORWARD, che rende la regola postata prima inefficace.

Ma non è tutto. Infatti, il demone Docker crea molte regole di iptables quando inizia a fare la sua magia riguardo alla connettività di rete dei container. In particolare, viene creata una tabella DOCKER per gestire le regole relative ai contenitori inoltrando il traffico dalla tabella FORWARD a questa nuova tabella.

```
$ iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy DROP)
target     prot opt source               destination
DOCKER-ISOLATION  all  --  anywhere             anywhere
DOCKER     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere             ctstate RELATED,ESTABLISHED
ACCEPT     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere
DOCKER     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere             ctstate RELATED,ESTABLISHED
ACCEPT     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

Chain DOCKER (2 references)
target     prot opt source               destination
ACCEPT     tcp  --  anywhere             172.18.0.4           tcp dpt:https
ACCEPT     tcp  --  anywhere             172.18.0.4           tcp dpt:http

Chain DOCKER-ISOLATION (1 references)
target     prot opt source               destination
DROP       all  --  anywhere             anywhere
DROP       all  --  anywhere             anywhere
RETURN     all  --  anywhere             anywhere
```

La soluzione

Se si controlla la documentazione ufficiale (https://docs.docker.com/v1.5/articles/networking/) , viene fornita una prima soluzione per limitare l'accesso al contenitore Docker a un particolare IP.

```
$ iptables -I DOCKER -i ext_if ! -s 8.8.8.8 -j DROP
```

In effetti, aggiungere una regola nella parte superiore della tabella DOCKER è una buona idea.

Non interferisce con le regole configurate automaticamente da Docker ed è semplice. Ma due grandi mancanze:

- Innanzitutto, cosa succede se è necessario accedere da due IP anziché uno? Qui può essere accettato solo un IP src, altri verranno eliminati senza alcun modo per impedirlo.
- In secondo luogo, cosa succede se la finestra mobile ha bisogno di accedere a Internet? Praticamente nessuna richiesta avrà successo, in quanto solo il server 8.8.8.8 potrebbe rispondere ad essi.
- Infine, cosa succede se si desidera aggiungere altre logiche? Ad esempio, concedere l'accesso a qualsiasi utente al server web che serve sul protocollo HTTP, ma limitare tutto il resto a particolari IP.

Per la prima osservazione, possiamo usare *ipset* . Invece di consentire un IP nella regola sopra, consentiamo tutti gli IP dall'ipset predefinito. Come bonus, l'ipset può essere aggiornato senza la necessità di ridefinire la regola iptable.

```
$ iptables -I DOCKER -i ext_if -m set ! --match-set my-ipset src -j DROP
```

Per la seconda osservazione, si tratta di un problema canonico per i firewall: se si è autorizzati a contattare un server attraverso un firewall, il firewall dovrebbe autorizzare il server a rispondere alla richiesta. Questo può essere fatto autorizzando pacchetti che sono collegati ad una connessione stabilita. Per la logica docker, fornisce:

```
$ iptables -I DOCKER -i ext_if -m state --state ESTABLISHED,RELATED -j ACCEPT
```

L'ultima osservazione si concentra su un punto: le regole di iptables sono essenziali. Infatti, la logica aggiuntiva per ACCETTARE alcune connessioni (inclusa quella relativa alle connessioni ESTABLISHED) deve essere posta nella parte superiore della tabella DOCKER, prima della regola DROP che nega tutte le connessioni rimanenti che non corrispondono all'ipset.

Poiché utilizziamo l'opzione -I di iptable, che inserisce le regole nella parte superiore della tabella, le precedenti regole di iptables devono essere inserite in ordine inverso:

```
// Drop rule for non matching IPs
$ iptables -I DOCKER -i ext_if -m set ! --match-set my-ipset src -j DROP
// Then Accept rules for established connections
$ iptables -I DOCKER -i ext_if -m state --state ESTABLISHED,RELATED -j ACCEPT
$ iptables -I DOCKER -i ext_if ... ACCEPT // Then 3rd custom accept rule
$ iptables -I DOCKER -i ext_if ... ACCEPT // Then 2nd custom accept rule
$ iptables -I DOCKER -i ext_if ... ACCEPT // Then 1st custom accept rule
```

Tenendo tutto questo a mente, ora puoi controllare gli esempi che illustrano questa configurazione.

Examples

Limita l'accesso ai contenitori Docker a un set di IP

Innanzitutto, installa *IPSET* se necessario. Si prega di fare riferimento alla vostra distribuzione per sapere come farlo. Ad esempio, ecco il comando per le distribuzioni di tipo Debian.

```
$ apt-get update
$ apt-get install ipset
```

Quindi creare un file di configurazione per definire un IPS contenente gli IP per i quali si desidera aprire l'accesso ai contenitori Docker.

```
$ vi /etc/ipfriends.conf
# Recreate the ipset if needed, and flush all entries
create -exist ipfriends hash:ip family inet hashsize 1024 maxelem 65536
flush
# Give access to specific ips
add ipfriends XXX.XXX.XXX.XXX
add ipfriends YYY.YYY.YYY.YYY
```

Carica questo ipset.

```
$ ipset restore < /etc/ipfriends.conf
```

Assicurati che il tuo demone Docker sia in esecuzione: non dovrebbe essere mostrato alcun errore dopo aver inserito il seguente comando.

```
$ docker ps
```

Sei pronto per inserire le tue regole di iptables. È **necessario** rispettare l'ordine.

```
// All requests of src ips not matching the ones from ipset ipfriends will be dropped.
$ iptables -I DOCKER -i ext_if -m set ! --match-set ipfriends src -j DROP
// Except for requests coming from a connection already established.
$ iptables -I DOCKER -i ext_if -m state --state ESTABLISHED,RELATED -j ACCEPT
```

Se vuoi creare nuove regole, dovrai rimuovere tutte le regole personalizzate che hai aggiunto prima di inserirne di nuove.

```
$ iptables -D DOCKER -i ext_if -m set ! --match-set ipfriends src -j DROP
$ iptables -D DOCKER -i ext_if -m state --state ESTABLISHED,RELATED -j ACCEPT
```

Configurare l'accesso alla restrizione all'avvio del daemon Docker

Lavori in corso

Alcune regole personalizzate di iptables

Lavori in corso

Leggi Iptables con Docker online: https://riptutorial.com/it/docker/topic/9201/iptables-con-docker

Capitolo 23: Ispezionando un contenitore funzionante

Sintassi

- finestra mobile ispeziona [OPZIONI] CONTENITORE | IMMAGINE [CONTENITORE | IMMAGINE ...]

Examples

Ottieni informazioni sul contenitore

Per ottenere tutte le informazioni per un contenitore puoi eseguire:

```
docker inspect <container>
```

Ottieni informazioni specifiche da un contenitore

Puoi ottenere informazioni specifiche da un contenitore eseguendo:

```
docker inspect -f '<format>' <container>
```

Ad esempio, puoi ottenere le impostazioni di rete eseguendo:

```
docker inspect -f '{{ .NetworkSettings }}' <container>
```

Puoi anche ottenere solo l'indirizzo IP:

```
docker inspect -f '{{ .NetworkSettings.IPAddress }}' <container>
```

Il parametro -f significa format e riceverà un modello Go come input per formattare ciò che è previsto, ma questo non porterà un bel ritorno, quindi prova:

```
docker inspect -f '{{ json .NetworkSettings }}' {{containerIdOrName}}
```

la parola chiave json restituirà il risultato come JSON.

Quindi per finire, un piccolo consiglio è usare python per formattare l'output JSON:

```
docker inspect -f '{{ json .NetworkSettings }}' <container> | python -mjson.tool
```

E voilà, puoi controllare qualsiasi cosa sul docker ispezionarlo e farlo sembrare carino nel tuo terminale.

E 'anche possibile utilizzare un'utility chiamata " jq " al fine di contribuire processo `docker inspect` l'output del comando.

```
docker inspect -f '{{ json .NetworkSettings }}' aa1 | jq [.Gateway]
```

Il comando precedente restituirà il seguente risultato:

```
[
  "172.17.0.1"
]
```

Questo output è in realtà una lista contenente un elemento. A volte, `docker inspect` mostra un elenco di diversi elementi e potresti voler fare riferimento a un elemento specifico. Ad esempio, se `Config.Env` contiene diversi elementi, puoi fare riferimento al primo elemento di questo elenco usando l' `index` :

```
docker inspect --format '{{ index (index .Config.Env) 0 }}' <container>
```

Il primo elemento è indicizzato a zero, il che significa che il secondo elemento di questa lista è all'indice `1` :

```
docker inspect --format '{{ index (index .Config.Env) 1 }}' <container>
```

Usando `len` è possibile ottenere il numero di elementi della lista:

```
docker inspect --format `{{ len .Config.Env }}` <container>
```

E usando numeri negativi, è possibile fare riferimento all'ultimo elemento della lista:

```
docker inspect -format "{{ index .Config.Cmd $[$(docker inspect -format `{{ len .Config.Cmd
}}` <container>)-1]}}" <container>
```

Alcune informazioni di `docker inspect` fornite come un dizionario di chiave: valore, ecco un estratto di un `docker inspect` un contenitore di jess / spotify in esecuzione

```
"Config": { "Hostname": "8255f4804dde", "Domainname": "", "User": "spotify", "AttachStdin":
false, "AttachStdout": false, "AttachStderr": false, "Tty": false, "OpenStdin": false,
"StdinOnce": false, "Env": [ "DISPLAY=unix:0",
"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin", "HOME=/home/spotify" ],
"Cmd": [ "-stylesheet=/home/spotify/spotify-override.css" ], "Image": "jess/spotify", "Volumes":
null, "WorkingDir": "/home/spotify", "Entrypoint": [ "spotify" ], "OnBuild": null, "Labels": {}
},
```

quindi prendo i valori dell'intera sezione Config

```
docker inspect -f '{{.Config}}' 825

{8255f4804dde spotify false false false map[] false false false [DISPLAY=unix:0
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin HOME=/home/spotify] [-
stylesheet=/home/spotify/spotify-override.css] false jess/spotify map[] /home/spotify [spotify]
false [] map[] }
```

ma anche un singolo campo, come il valore di Config.Image

```
docker inspect -f '{{index (.Config) "Image" }}' 825

jess/spotify
```

o Config.Cmd

```
docker inspect -f '{{.Config.Cmd}}' 825

[-stylesheet=/home/spotify/spotify-override.css]
```

Ispeziona un'immagine

Per ispezionare un'immagine, è possibile utilizzare l'ID immagine o il nome dell'immagine, costituito da repository e tag. Di ', hai l'immagine base di CentOS 6:

```
➜ ~ docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
centos centos6 cf2c3ece5e41 2 weeks ago 194.6 MB
```

In questo caso è possibile eseguire una delle seguenti operazioni:

- ➜ ~ docker inspect cf2c3ece5e41
- ➜ ~ docker inspect centos:centos6

Entrambi questi comandi ti daranno tutte le informazioni disponibili in un array JSON:

```
[
    {
        "Id": "sha256:cf2c3ece5e418fd063bfad5e7e8d083182195152f90aac3a5ca4dbfbf6a1fc2a",
        "RepoTags": [
            "centos:centos6"
        ],
        "RepoDigests": [],
        "Parent": "",
        "Comment": "",
        "Created": "2016-07-01T22:34:39.970264448Z",
        "Container": "b355fe9a01a8f95072e4406763138c5ad9ca0a50dbb0ce07387ba905817d6702",
        "ContainerConfig": {
            "Hostname": "68a1f3cfce80",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) CMD [\"/bin/bash\"]"
            ],
```

```
            "Image":
"sha256:cdbcc7980b002dc19b4d5b6ac450993c478927f673339b4e6893647fe2158fa7",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "build-date": "20160701",
                "license": "GPLv2",
                "name": "CentOS Base Image",
                "vendor": "CentOS"
            }
        },
        "DockerVersion": "1.10.3",
        "Author": "https://github.com/CentOS/sig-cloud-instance-images",
        "Config": {
            "Hostname": "68a1f3cfce80",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/bash"
            ],
            "Image":
"sha256:cdbcc7980b002dc19b4d5b6ac450993c478927f673339b4e6893647fe2158fa7",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "build-date": "20160701",
                "license": "GPLv2",
                "name": "CentOS Base Image",
                "vendor": "CentOS"
            }
        },
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 194606575,
        "VirtualSize": 194606575,
        "GraphDriver": {
            "Name": "aufs",
            "Data": null
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:2714f4a6cdee9d4c987fef019608a4f61f1cda7ccf423aeb8d7d89f745c58b18"
            ]
        }
    }
]
```

Stampa di informazioni specifiche

finestra mobile `inspect` supporti Vai modelli tramite l' `--format` opzione. Ciò consente una migliore integrazione negli script, senza ricorrere agli strumenti pipe / sed / grep tradizionali.

Stampa un IP interno del contenitore :

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' 7786807d8084
```

Questo è utile per l'accesso diretto alla rete della configurazione automatica dei bilanciatori di carico.

Stampa un PID di *inizializzazione* contenitore :

```
docker inspect --format '{{ .State.Pid }}' 7786807d8084
```

Questo è utile per un'ispezione più approfondita tramite `/proc` o strumenti come `strace` .

Formatura avanzata :

```
docker inspect --format 'Container {{ .Name }} listens on {{ .NetworkSettings.IPAddress }}:{{
range $index, $elem := .Config.ExposedPorts }}{{ $index }}{{ end }}' 5765847de886 7786807d8084
```

Produrrà:

```
Container /redis listens on 172.17.0.3:6379/tcp
Container /api listens on 172.17.0.2:4000/tcp
```

Eseguire il debug dei registri del contenitore utilizzando la finestra mobile

`docker inspect` comando `docker inspect` può essere utilizzato per eseguire il debug dei log del contenitore.

Lo stdout e lo stderr del contenitore possono essere controllati per eseguire il debug del contenitore, la cui ubicazione può essere ottenuta utilizzando il `docker inspect` .

Comando: `docker inspect <container-id> | grep Source`

Dà la posizione dei contenitori stdout e stderr.

Esaminare stdout / stderr di un contenitore in esecuzione

```
docker logs --follow <containerid>
```

Questo tails l'output del contenitore in esecuzione. Ciò è utile se non hai impostato un driver di registrazione sul daemon docker.

Leggi Ispezionando un contenitore funzionante online:

https://riptutorial.com/it/docker/topic/1336/ispezionando-un-contenitore-funzionante

Capitolo 24: Limitazione dell'accesso alla rete del contenitore

Osservazioni

Esempio di reti mobili che bloccano il traffico. Utilizzare come rete all'avvio del contenitore con `--net` o `docker network connect --net` .

Examples

Blocca l'accesso alla LAN e fuori

```
docker network create -o "com.docker.network.bridge.enable_ip_masquerade"="false" lan-restricted
```

- blocchi
  - LAN locale
  - Internet
- Non blocca
  - Host eseguendo daemon docker (esempio di accesso a `10.0.1.10:22`)

Blocca l'accesso ad altri contenitori

```
docker network create -o "com.docker.network.bridge.enable_icc"="false" icc-restricted
```

- blocchi
  - Contenitori che accedono ad altri contenitori sulla stessa rete con `icc-restricted` .
- Non blocca
  - Accesso al daemon della docker in esecuzione host
  - LAN locale
  - Internet

Bloccare l'accesso dai contenitori all'host locale che esegue il daemon della finestra mobile

```
iptables -I INPUT -i docker0 -m addrtype --dst-type LOCAL -j DROP
```

- blocchi
  - Accesso al daemon della docker in esecuzione host
- Non blocca
  - Traffico da container a container
  - LAN locale
  - Internet

- Reti mobili personalizzate che non utilizzano `docker0`

Bloccare l'accesso dai contenitori all'host locale che esegue il daemon docker (rete personalizzata)

```
docker network create --subnet=192.168.0.0/24 --gateway=192.168.0.1 --ip-range=192.168.0.0/25
local-host-restricted
iptables -I INPUT -s 192.168.0.0/24 -m addrtype --dst-type LOCAL -j DROP
```

Crea una rete chiamata `local-host-restricted` che quale:

- blocchi
  - Accesso al daemon della docker in esecuzione host
- Non blocca
  - Traffico da container a container
  - LAN locale
  - Internet
  - Accesso proveniente da altre reti mobili

Le reti personalizzate hanno nomi di bridge come `br-15bbe9bb5bf5` , quindi usiamo invece la sua subnet.

Leggi Limitazione dell'accesso alla rete del contenitore online:
https://riptutorial.com/it/docker/topic/6331/limitazione-dell-accesso-alla-rete-del-contenitore

Capitolo 25: Modalità Docker --net (bridge, hots, contenitore mappato e nessuno).

introduzione

Iniziare

Modalità Bridge È un default e collegato al bridge docker0. Metti il contenitore su un namespace di rete completamente separato.

Modalità host Quando il contenitore è solo un processo in esecuzione in un host, il contenitore verrà collegato alla scheda NIC host.

Modalità contenitore mappato Questa modalità essenzialmente associa un nuovo contenitore a uno stack di rete di contenitori esistente. Viene anche chiamato "contenitore in modalità contenitore".

Nessuno Indica al docker di mettere il contenitore nel proprio stack di rete senza configurazione

Examples

Modalità Bridge, modalità host e modalità contenitore mappato

Modalità Ponte

```
$ docker run -d --name my_app -p 10000:80 image_name
```

Si noti che non è stato necessario specificare **--net = bridge** perché questa è la modalità di lavoro predefinita per la finestra mobile. Ciò consente di eseguire più contenitori per l'esecuzione sullo stesso host senza alcuna assegnazione di porta dinamica. Quindi la modalità **BRIDGE** evita il conflitto tra le porte ed è sicuro poiché ogni contenitore esegue il proprio spazio dei nomi di rete privata.

Modalità host

```
$ docker run -d --name my_app -net=host image_name
```

Poiché utilizza lo spazio dei nomi della rete host, non è necessaria la configurazione speciale ma può comportare problemi di sicurezza.

Modalità contenitore mappato

Questa modalità mappa essenzialmente un nuovo contenitore in uno stack di rete dei contenitori esistente. Ciò implica che le risorse di rete come l'indirizzo IP e le mappature delle porte del primo contenitore saranno condivise dal secondo contenitore. Questo è anche chiamato modalità

'contenitore nel contenitore'. Supponiamo che tu abbia due contatti come web_container_1 e web_container_2 e eseguiremo web_container_2 in modalità contenitore mappato. Scarichiamo innanzitutto web_container_1 e lo eseguiamo in modalità distaccata con il seguente comando,

```
$ docker run -d --name web1 -p 80:80 USERNAME/web_container_1
```

Una volta scaricato, diamo un'occhiata e assicuriamoci che funzioni. Qui abbiamo appena mappato una porta in un contenitore in esecuzione nella modalità bridge predefinita. Ora, eseguiamo un secondo contenitore in modalità contenitore mappato. Lo faremo con questo comando.

```
$ docker run -d --name web2 --net=container:web1 USERNAME/web_container_2
```

Ora, se si ottengono semplicemente le informazioni di interfaccia su entrambi i contatori, si otterrà la stessa configurazione di rete. Questo include in realtà la modalità HOST che mappa con le informazioni esatte dell'host. Il primo contaienr è stato eseguito in modalità bridge predefinita e il secondo contenitore è in esecuzione in modalità contenitore mappato. Possiamo ottenere risultati molto simili avviando il primo contenitore in modalità host e il secondo contenitore in modalità contenitore mappato.

Leggi Modalità Docker --net (bridge, hots, contenitore mappato e nessuno). online: https://riptutorial.com/it/docker/topic/9643/modalita-docker---net--bridge--hots--contenitore-mappato-e-nessuno--

Capitolo 26: Modalità sciame Docker

introduzione

Uno sciame è un numero di Docker Engines (o *nodi*) che distribuiscono *servizi* collettivamente. Swarm viene utilizzato per distribuire l'elaborazione su molte macchine fisiche, virtuali o cloud.

Sintassi

- Inizializzazione di uno sciame : finestra mobile sciame init [OPZIONI]
- Unisciti a uno sciame come nodo e / o gestore : join sciame docker [OPZIONI] HOST: PORT
- Creare un nuovo servizio : creazione di una finestra mobile [OPZIONI] IMMAGINE [COMANDO] [ARG ...]
- Visualizza informazioni dettagliate su uno o più servizi : ispezioni servizio docker [OPZIONI] SERVIZIO [SERVIZIO ...]
- Elenco servizi : servizio docker ls [OPZIONI]
- Rimuovere uno o più servizi : servizio docker rm SERVICE [SERVICE ...]
- Scalare uno o più servizi replicati : scala del servizio finestra mobile SERVICE = REPLICAS [SERVICE = REPLICAS ...]
- Elencare le attività di uno o più servizi : docker service ps [OPZIONI] SERVICE [SERVICE ...]
- Aggiornamento di un servizio : aggiornamento del servizio finestra mobile [OPZIONI] SERVIZIO

Osservazioni

La modalità Sciame implementa le seguenti funzionalità:

- Gestione dei cluster integrata con Docker Engine
- Design decentralizzato
- Modello di servizio dichiarativo
- scalata
- Riconciliazione di stato desiderata
- Rete multi-host
- Scoperta del servizio
- Bilancio del carico
- Design sicuro per impostazione predefinita
- Aggiornamenti a rotazione

Per ulteriori informazioni sulla documentazione ufficiale di Docker in merito alla visita di Swarm : panoramica sulla modalità Swarm

Comandi CLI della modalità scia

Fare clic sulla descrizione dei comandi per la documentazione

Inizializza uno sciame

```
docker swarm init [OPTIONS]
```

Unisciti a uno sciame come nodo e / o manager

```
docker swarm join [OPTIONS] HOST:PORT
```

Crea un nuovo servizio

```
docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Visualizza informazioni dettagliate su uno o più servizi

```
docker service inspect [OPTIONS] SERVICE [SERVICE...]
```

Elenco dei servizi

```
docker service ls [OPTIONS]
```

Rimuovere uno o più servizi

```
docker service rm SERVICE [SERVICE...]
```

Scala uno o più servizi replicati

```
docker service scale SERVICE=REPLICAS [SERVICE=REPLICAS...]
```

Elencare le attività di uno o più servizi

```
docker service ps [OPTIONS] SERVICE [SERVICE...]
```

Aggiorna un servizio

```
docker service update [OPTIONS] SERVICE
```

Examples

Crea uno sciame su Linux usando docker-machine e VirtualBox

```
# Create the nodes
# In a real world scenario we would use at least 3 managers to cover the fail of one manager.
docker-machine create -d virtualbox manager
docker-machine create -d virtualbox worker1

# Create the swarm
# It is possible to define a port for the *advertise-addr* and *listen-addr*, if none is
defined the default port 2377 will be used.
docker-machine ssh manager \
    docker swarm init \
    --advertise-addr $(docker-machine ip manager)
    --listen-addr $(docker-machine ip manager)

# Extract the Tokens for joining the Swarm
# There are 2 different Tokens for joining the swarm.
MANAGER_TOKEN=$(docker-machine ssh manager docker swarm join-token manager --quiet)
WORKER_TOKEN=$(docker-machine ssh manager docker swarm join-token worker --quiet)


# Join a worker node with the worker token
docker-machine ssh worker1 \
    docker swarm join \
    --token $WORKER_TOKEN \
    --listen-addr $(docker-machine ip worker1) \
    $(docker-machine ip manager):2377
```

Scopri i token di join di worker e manager

Quando si automatizza il provisioning di nuovi nodi su uno sciame, è necessario sapere qual è il giusto token di join sia per lo sciame che per l'indirizzo pubblicizzato del gestore. Puoi scoprirlo eseguendo i seguenti comandi su uno dei nodi del gestore esistente:

```
# grab the ipaddress:port of the manager (second last line minus the whitespace)
export MANAGER_ADDRESS=$(docker swarm join-token worker | tail -n 2 |  tr -d '[[:space:]]')

# grab the manager and worker token
export MANAGER_TOKEN=$(docker swarm join-token manager -q)
export WORKER_TOKEN=$(docker swarm join-token worker -q)
```

L'opzione -q emette solo il token. Senza questa opzione si ottiene il comando completo per la registrazione a uno sciame.

Quindi su nodi appena sottoposti a provisioning, puoi unirti allo sciame usando.

```
docker swarm join --token $WORKER_TOKEN $MANAGER_ADDRESS
```

Ciao domanda mondiale

Di solito vorresti creare una serie di servizi per formare un'applicazione replicata e orchestrata.

Una tipica applicazione Web moderna è costituita da un database, api, frontend e reverse proxy.

Persistenza

Il database ha bisogno di persistenza, quindi abbiamo bisogno di un filesystem condiviso su tutti i nodi di uno sciame. Può essere NAS, server NFS, GFS2 o qualsiasi altra cosa. L'impostazione è fuori portata qui. Attualmente Docker non contiene e non gestisce la persistenza in uno sciame. Questo esempio presuppone che ci sia `/nfs/` posizione condivisa montata su tutti i nodi.

Rete

Per essere in grado di comunicare tra loro, i servizi in uno sciame devono essere sulla stessa rete.

Scegli un intervallo IP (qui `10.0.9.0/24`) e il nome della rete (`hello-network`) ed esegui un comando:

```
docker network create \
  --driver overlay \
  --subnet 10.0.9.0/24 \
  --opt encrypted \
  hello-network
```

Banca dati

Il primo servizio di cui abbiamo bisogno è un database. Usiamo postgresql come esempio. Crea una cartella per un database in `nfs/postgres` ed esegui questo:

```
docker service create --replicas 1 --name hello-db \
       --network hello-network -e PGDATA=/var/lib/postgresql/data \
       --mount type=bind,src=/nfs/postgres,dst=/var/lib/postgresql/data \
       kiasaki/alpine-postgres:9.5
```

Nota che abbiamo usato le `--network hello-network` e `--mount` .

API

La creazione dell'API non rientra nell'ambito di questo esempio, quindi facciamo finta di avere un'immagine API sotto `username/hello-api` .

```
docker service create --replicas 1 --name hello-api \
       --network hello-network \
       -e NODE_ENV=production -e PORT=80 -e POSTGRESQL_HOST=hello-db \
       username/hello-api
```

Si noti che abbiamo passato un nome al nostro servizio di database. Docker swarm ha un server DNS round robin incorporato, quindi l'API sarà in grado di connettersi al database usando il suo nome DNS.

Reverse proxy

Creiamo il servizio nginx per servire la nostra API in un mondo esterno. Creare i file di configurazione di nginx in un percorso condiviso ed eseguire questo:

```
docker service create --replicas 1 --name hello-load-balancer \
       --network hello-network \
       --mount type=bind,src=/nfs/nginx/nginx.conf,dst=/etc/nginx/nginx.conf \
       -p 80:80 \
       nginx:1.10-alpine
```

Si noti che abbiamo usato l'opzione `-p` per pubblicare una porta. Questa porta sarebbe disponibile per qualsiasi nodo in uno sciame.

Nodo Disponibilità

Disponibilità del nodo modalità sciame:

- Attivo significa che lo schedulatore può assegnare compiti a un nodo.
- Pausa indica che lo scheduler non assegna nuove attività al nodo, ma le attività esistenti rimangono in esecuzione.
- Scarico indica che lo scheduler non assegna nuove attività al nodo. Lo scheduler chiude tutte le attività esistenti e le pianifica su un nodo disponibile.

Per cambiare la disponibilità della modalità:

```
#Following commands can be used on swarm manager(s)
docker node update --availability drain node-1
#to verify:
docker node ls
```

Promuovi o abbassa i nodi dello sciame

Per promuovere un nodo o un insieme di nodi, eseguire il `docker node promote` da un nodo gestore:

```
docker node promote node-3 node-2

Node node-3 promoted to a manager in the swarm.
Node node-2 promoted to a manager in the swarm.
```

Per abbassare di livello un nodo o un set di nodi, eseguire il `docker node demote` da un nodo gestore:

```
docker node demote node-3 node-2

Manager node-3 demoted in the swarm.
Manager node-2 demoted in the swarm.
```

Lasciando lo Sciame

Nodo lavoratore:

```
#Run the following on the worker node to leave the swarm.

docker swarm leave
Node left the swarm.
```

Se il nodo ha il ruolo *Manager* , riceverai un avviso sul mantenimento del quorum dei Manager. Puoi usare --force per lasciare sul nodo manager:

```
#Manager Node

docker swarm leave --force
Node left the swarm.
```

I nodi che hanno lasciato lo Swarm verranno comunque visualizzati nell'output del `docker node ls` .

Per rimuovere i nodi dall'elenco:

```
docker node rm node-2

node-2
```

Leggi Modalità sciame Docker online: https://riptutorial.com/it/docker/topic/749/modalita-sciame-docker

Capitolo 27: Ordinamento contenuti Dockerfile

Osservazioni

1. Dichiarazione immagine di base (`FROM`)
2. Metadati (ad es. `MAINTAINER` , `LABEL`)
3. Installazione delle dipendenze del sistema (ad es. `apt-get install` , `apk add`)
4. Copia del file delle dipendenze dell'app (ad es. `bower.json` , `package.json` , `build.gradle` , `requirements.txt`)
5. Installazione delle dipendenze dell'app (ad esempio `npm install` , `pip install`)
6. Copia di tutta la base di codice
7. Impostazione configs runtime di default (ad esempio `CMD` , `ENTRYPOINT` , `ENV` , `EXPOSE`)

Questi ordini sono fatti per ottimizzare i tempi di costruzione utilizzando il meccanismo di memorizzazione nella cache incorporato di Docker.

Regola generale:

> Parti che cambiano spesso (ad es. Codebase) dovrebbero essere posizionate vicino al fondo del Dockerfile e viceversa. Le parti che cambiano raramente (ad es. Dipendenze) dovrebbero essere posizionate in alto.

Examples

Dockerfile semplice

```
# Base image
FROM python:2.7-alpine

# Metadata
MAINTAINER John Doe <johndoe@example.com>

# System-level dependencies
RUN apk add --update \
    ca-certificates \
    && update-ca-certificates \
    && rm -rf /var/cache/apk/*

# App dependencies
COPY requirements.txt /requirements.txt
RUN pip install -r /requirements.txt

# App codebase
WORKDIR /app
COPY . ./

# Configs
ENV DEBUG true
```

```
EXPOSE 5000
CMD ["python", "app.py"]
```

MAINTAINER sarà deprecato in Docker 1.13 e dovrebbe essere sostituito utilizzando LABEL. (Fonte)

Esempio: LABEL Maintainer = "John Doe johndoe@example.com"

Leggi Ordinamento contenuti Dockerfile online:
https://riptutorial.com/it/docker/topic/6448/ordinamento-contenuti-dockerfile

Capitolo 28: passaggio di dati segreti a un contenitore in esecuzione

Examples

modi per passare i segreti in un contenitore

Il modo non molto sicuro (perché la `docker inspect` lo mostrerà) è il passaggio a una variabile di ambiente

```
docker run
```

ad esempio

```
docker run -e password=abc
```

o in un file

```
docker run --env-file myfile
```

dove myfile può contenere

```
password1=abc password2=def
```

è anche possibile inserirli in un volume

```
docker run -v $(pwd)/my-secret-file:/secret-file
```

alcuni modi migliori, usare

keywhiz https://square.github.io/keywhiz/

vault https://www.hashicorp.com/blog/vault.html

etcd con crypt https://xordataexchange.github.io/crypt/

Leggi passaggio di dati segreti a un contenitore in esecuzione online: https://riptutorial.com/it/docker/topic/6481/passaggio-di-dati-segreti-a-un-contenitore-in-esecuzione

Capitolo 29: Più processi in un'istanza contenitore

Osservazioni

Di solito ogni contenitore dovrebbe ospitare un processo. Nel caso in cui siano necessari più processi in un contenitore (ad esempio un server SSH per accedere all'istanza del contenitore in esecuzione) è possibile ottenere l'idea di scrivere il proprio script della shell che avvia tali processi. In tal caso, dovresti occuparti della gestione del `SIGNAL` (ad esempio, reindirizzare un `SIGINT` catturato ai processi figli del tuo script). Non è proprio quello che vuoi. Una soluzione semplice consiste nell'utilizzare `supervisord` come processo root dei contenitori che si occupa della gestione `SIGNAL` e della durata dei processi figli.

Ma tieni presente che questa non è la "via mobile". Per ottenere questo esempio in modalità docker, accedere `docker host` (la macchina su cui è in esecuzione il contenitore) ed eseguire la `docker exec -it container_name /bin/bahs` . Questo comando ti apre una shell all'interno del contenitore come farebbe ssh.

Examples

Dockerfile + supervisord.conf

Per eseguire più processi, ad esempio un server Web Apache insieme a un daemon SSH all'interno dello stesso contenitore, è possibile utilizzare `supervisord` .

Crea il tuo file di configurazione `supervisord.conf` come:

```
[supervisord]
nodaemon=true

[program:sshd]
command=/usr/sbin/sshd -D

[program:apache2]
command=/bin/bash -c "source /etc/apache2/envvars && exec /usr/sbin/apache2 -DFOREGROUND"
```

Quindi creare un file `Dockerfile` come:

```
FROM ubuntu:16.04
RUN apt-get install -y openssh-server apache2 supervisor
RUN mkdir -p /var/lock/apache2 /var/run/apache2 /var/run/sshd /var/log/supervisor
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
CMD ["/usr/bin/supervisord"]
```

Quindi puoi costruire la tua immagine:

```
docker build -t supervisord-test .
```

Successivamente puoi eseguirlo:

```
$ docker run -p 22 -p 80 -t -i supervisord-test
2016-07-26 13:15:21,101 CRIT Supervisor running as root (no user in config file)
2016-07-26 13:15:21,101 WARN Included extra file     "/etc/supervisor/conf.d/supervisord.conf"
during parsing
2016-07-26 13:15:21,112 INFO supervisord started with pid 1
2016-07-26 13:15:21,113 INFO spawned: 'sshd' with pid 6
2016-07-26 13:15:21,115 INFO spawned: 'apache2' with pid 7
...
```

Leggi Più processi in un'istanza contenitore online: https://riptutorial.com/it/docker/topic/4053/piu-processi-in-un-istanza-contenitore

Capitolo 30: Registrazione

Examples

Configurazione di un driver di registro nel servizio systemd

```
[Service]

# empty exec prevents error "docker.service has more than one ExecStart= setting, which is
only allowed for Type=oneshot services. Refusing."
ExecStart=
ExecStart=/usr/bin/dockerd -H fd:// --log-driver=syslog
```

Ciò abilita la registrazione syslog per il daemon docker. Il file dovrebbe essere creato nella directory appropriata con il proprietario root, che in genere sarebbe `/etc/systemd/system/docker.service.d` per esempio su Ubuntu 16.04.

Panoramica

L'approccio Docker alla registrazione è che si costruiscono i contenitori in modo tale che i registri vengano scritti sullo standard output (console / terminale).

Se hai già un contenitore che scrive i log in un file, puoi reindirizzare creando un link simbolico:

```
ln -sf /dev/stdout /var/log/nginx/access.log
ln -sf /dev/stderr /var/log/nginx/error.log
```

Dopo averlo fatto, puoi usare vari driver di registro per mettere i tuoi log dove ti servono.

Leggi Registrazione online: https://riptutorial.com/it/docker/topic/7378/registrazione

Capitolo 31: Registro di sistema privato / sicuro Docker con API v2

introduzione

Un registro docker privato e sicuro invece di un Docker Hub. Sono richieste le abilità di docker di base.

Parametri

Comando	Spiegazione
sudo docker run -p 5000: 5000	Avviare un contenitore di finestra mobile e collegare la porta 5000 dal contenitore alla porta 5000 della macchina fisica.
--nome registro	Nome del contenitore (utilizzare per migliorare la leggibilità di "docker ps").
-v 'pwd' / certs: / certs	Collega CURRENT_DIR / certs della macchina fisica su / certs del contenitore (come una "cartella condivisa").
-e REGISTRY_HTTP_TLS_CERTIFICATE = / certs / server.crt	Specifichiamo che il registro dovrebbe usare il file /certs/server.crt per iniziare. (variabile env)
-e REGISTRY_HTTP_TLS_KEY = / certs / server.key	Lo stesso per la chiave RSA (server.key).
-v / root / images: / var / lib / registry /	Se si desidera salvare tutte le immagini del registro, è necessario farlo sul computer fisico. Qui salviamo tutte le immagini su / root / images sulla macchina fisica. Se si esegue questa operazione, è possibile interrompere e riavviare il registro senza perdere alcuna immagine.
Registro di sistema: 2	Specifichiamo che vorremmo estrarre l'immagine del registro dall'hub docker (o localmente), e aggiungiamo «2» perché vogliamo installare la versione 2 del registro.

Osservazioni

Come installare un motore mobile (chiamato client in questo tutorial)

Come generare un certificato autofirmato SSL

Examples

Generazione di certificati

Genera una chiave privata RSA: `openssl genrsa -des3 -out server.key 4096`

Openssl dovrebbe chiedere una passphrase in questo passaggio. Si noti che verrà utilizzato solo il certificato per la comunicazione e l'autenticazione, senza passphrase. Basta usare 123456 per esempio.

Genera la richiesta di firma del certificato: `openssl req -new -key server.key -out server.csr`

Questo passaggio è importante perché ti verranno chieste alcune informazioni sui certificati. Le informazioni più importanti sono "Nome comune", ovvero il nome del dominio, che può essere utilizzato per le comunicazioni tra il registro di posta elettronica privato e tutte le altre macchine. Esempio: mydomain.com

Rimuovi la passphrase dalla chiave privata RSA: `cp server.key server.key.org && openssl rsa -in server.key.org -out server.key`

Come ho detto, ci concentreremo sul certificato senza passaparola. Quindi fai attenzione con tutti i file della tua chiave (.key, .csr, .crt) e conservali in un posto sicuro.

Genera il certificato autofirmato: `openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt`

Sono ora disponibili due file essenziali, *server.key* e *server.crt* , necessari per l'autenticazione del registro privato.

Esegui il registro con un certificato autofirmato

Per eseguire il registro privato (in modo sicuro) è necessario generare un certificato autofirmato, è possibile fare riferimento all'esempio precedente per generarlo.

Per il mio esempio ho messo *server.key* e *server.crt* in / root / certs

Prima di eseguire il comando finestra mobile devi essere posizionato (usa `cd`) nella directory che contiene la cartella *certs* . Se non lo sei e provi a eseguire il comando riceverai un errore come

> level = fatal msg = "apri /certs/server.crt: nessun file o directory"

Quando sei (`cd /root` nel mio esempio), puoi fondamentalmente avviare il registro sicuro / privato usando: `sudo docker run -p 5000:5000 --restart=always --name registry -v `pwd`/certs:/certs -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/server.crt -e REGISTRY_HTTP_TLS_KEY=/certs/server.key -v /root/Documents:/var/lib/registry/ registry:2`
Spiegazioni sul comando sono disponibili sulla parte Parametri.

Tirare o spingere da un client finestra mobile

Quando si ottiene un registro di lavoro in esecuzione, è possibile trascinare o spingere le immagini su di esso. Per questo è necessario il file *server.crt* in una cartella speciale sul client docker. Il certificato consente di autenticare con il registro e quindi crittografare la comunicazione.

Copia *server.crt* dalla macchina del registro in /etc/docker/certs.d/mydomain.com:5000/ sul computer client. Quindi rinominarlo in *ca-certificates.crt* : `mv /etc/docker/certs.d/mydomain.com:5000/server.crt /etc/docker/certs.d/mydomain.com:5000/ca-certificates.crt`

A questo punto puoi tirare o spingere le immagini dal tuo registro privato:
PULL: `docker pull mydomain.com:5000/nginx` o
SPINGERE :

1. Ottieni un'immagine ufficiale da hub.docker.com: docker `docker pull nginx`
2. Contrassegna questa immagine prima di passare al registro privato: `docker tag IMAGE_ID mydomain.com:5000/nginx` (usa le `docker images` per ottenere IMAGE_ID)
3. Spingere l'immagine nel registro: `docker push mydomain.com:5000/nginx`

Leggi Registro di sistema privato / sicuro Docker con API v2 online:
https://riptutorial.com/it/docker/topic/8707/registro-di-sistema-privato---sicuro-docker-con-api-v2

Capitolo 32: Registro Docker

Examples

Esecuzione del registro

Non usare il `registry:latest` ! Questa immagine punta al vecchio registro v1. Questo progetto Python non viene più sviluppato. Il nuovo registro v2 è scritto in Go e viene mantenuto attivamente. Quando le persone si riferiscono a un "registro privato" si riferiscono al registro v2, *non* al registro v1!

```
docker run -d -p 5000:5000 --name="registry" registry:2
```

Il comando precedente esegue la versione più recente del registro, che può essere trovata nel progetto Docker Distribution .

Per ulteriori esempi di funzionalità di gestione delle immagini, come tagging, pull o push, vedere la sezione sulla gestione delle immagini.

Configura il registro con il back-end di archiviazione di AWS S3

La configurazione di un registro privato per l'utilizzo di un back-end AWS S3 è semplice. Il registro può farlo automaticamente con la configurazione corretta. Ecco un esempio di cosa dovrebbe essere nel file `config.yml` :

```
storage:
    s3:
        accesskey: AKAAAAAACCCCCCCBBBDA
        secretkey: rn9rjnNuX44iK+26qpM4cDEoOnonbBW98FYaiDtS
        region: us-east-1
        bucket: registry.example.com
        encrypt: false
        secure: true
        v4auth: true
        chunksize: 5242880
        rootdirectory: /registry
```

I campi `accesskey` e `secretkey` sono credenziali IAM con autorizzazioni S3 specifiche (consultare la documentazione per ulteriori informazioni). Può altrettanto facilmente utilizzare le credenziali con la politica di `AmazonS3FullAccess` allegata. La `region` è la regione del tuo secchio S3. Il `bucket` è il nome del secchio. Puoi scegliere di archiviare le tue immagini crittografate con `encrypt` . Il campo `secure` indica l'uso di HTTPS. In generale, dovresti impostare `v4auth` su true, anche se il suo valore predefinito è false. Il campo `chunksize` ti consente di rispettare il requisito dell'S3 S3 secondo cui i caricamenti a blocchi hanno una dimensione di almeno cinque megabyte. Infine, `rootdirectory` specifica una directory sotto il tuo bucket S3 da utilizzare.

Esistono altri back-end di archiviazione che possono essere configurati altrettanto facilmente.

Leggi Registro Docker online: https://riptutorial.com/it/docker/topic/4173/registro-docker

Capitolo 33: Servizi in corso

Examples

Creare un servizio più avanzato

Nell'esempio seguente creeremo un servizio con il *visualizzatore* del nome. Specificheremo un'etichetta personalizzata e rimappare la porta interna del servizio da 8080 a 9090. Inoltre, vincoleremo il montaggio di una directory esterna dell'host nel servizio.

```
docker service create \
        --name=visualizer \
        --label com.my.custom.label=visualizer \
        --publish=9090:8080 \
        --mount type=bind,source=/var/run/docker.sock,target=/var/run/docker.sock \
        manomarks/visualizer:latest
```

Creare un servizio semplice

Questo semplice exampe creerà un servizio web mondiale Hello che ascolterà sulla porta 80.

```
docker service create \
    --publish 80:80 \
    tutum/hello-world
```

Rimozione di un servizio

Questo semplice esempio rimuoverà il servizio con il nome "visualizzatore":

```
docker service rm visualizer
```

Ridimensionamento di un servizio

Questo esempio ridimensiona il servizio a 4 istanze:

```
docker service scale visualizer=4
```

In modalità Swarm Docker non interrompiamo un servizio. Lo ridimensioniamo a zero:

```
docker service scale visualizer=0
```

Leggi Servizi in corso online: https://riptutorial.com/it/docker/topic/8802/servizi-in-corso

Capitolo 34: sicurezza

introduzione

Per mantenere aggiornate le nostre immagini per le patch di sicurezza, dobbiamo sapere da quale immagine di base dipendiamo

Examples

Come trovare da quale immagine proviene la nostra immagine

Ad esempio, vediamo un container Wordpress

Il Dockerfile inizia con FROM php: 5.6-apache

quindi andiamo al Dockerfile sopra menzionato https://github.com/docker-library/php/blob/master/5.6/apache/Dockerfile

e troviamo FROM debian: jessie Quindi questo significa che una patch di sicurezza appare per Debian jessie, abbiamo bisogno di ricostruire la nostra immagine.

Leggi sicurezza online: https://riptutorial.com/it/docker/topic/8077/sicurezza

Capitolo 35: Statistiche Docker tutti i contenitori in esecuzione

Examples

Statistiche Docker tutti i contenitori in esecuzione

```
sudo docker stats $(sudo docker inspect -f "{{ .Name }}" $(sudo docker ps -q))
```

Mostra l'utilizzo della CPU in tempo reale di tutti i contenitori in esecuzione.

Leggi Statistiche Docker tutti i contenitori in esecuzione online:
https://riptutorial.com/it/docker/topic/5863/statistiche-docker-tutti-i-contenitori-in-esecuzione

Capitolo 36: Volumi dati Docker

introduzione

I volumi di dati del Docker forniscono un modo per mantenere i dati indipendentemente dal ciclo di vita di un container. I volumi presentano una serie di funzioni utili come:

Montare una directory host all'interno del contenitore, condividendo i dati tra i contenitori usando il filesystem e preservando i dati se un container viene cancellato

Sintassi

- volume finestra mobile [OPZIONI] [COMMAND]

Examples

Montare una directory dall'host locale in un contenitore

È possibile montare una directory host su un percorso specifico nel contenitore usando l'opzione `-v` o `--volume` riga di comando. L'esempio seguente monterà `/etc` sull'host su `/mnt/etc` nel contenitore:

```
(on linux) docker run -v "/etc:/mnt/etc" alpine cat /mnt/etc/passwd
(on windows)  docker run -v "/c/etc:/mnt/etc" alpine cat /mnt/etc/passwd
```

L'accesso predefinito al volume all'interno del contenitore è di lettura-scrittura. Per montare un volume di sola lettura all'interno di un contenitore, utilizzare il suffisso `:ro` :

```
docker run -v "/etc:/mnt/etc:ro" alpine touch /mnt/etc/passwd
```

Creare un volume con nome

```
docker volume create --name="myAwesomeApp"
```

L'utilizzo di un volume denominato rende la gestione dei volumi molto più leggibile. È possibile creare un volume denominato utilizzando il comando specificato sopra, ma è anche possibile creare un volume denominato all'interno di un comando di `docker run` utilizzando l'opzione della riga di comando `-v` o `--volume` :

```
docker run -d --name="myApp-1" -v="myAwesomeApp:/data/app" myApp:1.5.3
```

Si noti che la creazione di un volume denominato in questo modulo è simile al montaggio di un file / directory host come volume, tranne per il fatto che anziché un percorso valido viene specificato il nome del volume. Una volta creati, i volumi denominati possono essere condivisi con altri

contenitori:

```
docker run -d --name="myApp-2" --volumes-from "myApp-1" myApp:1.5.3
```

Dopo aver eseguito il comando precedente, un nuovo contenitore è stato creato con il nome `myApp-2` dal `myApp:1.5.3` immagine, che condivide la `myAwesomeApp` nome volume con `myApp-1` . Il volume denominato `myAwesomeApp` è montato su `/data/app` nel contenitore `myApp-2` , così come è montato su `/data/app` nel contenitore `myApp-1` .

Leggi Volumi dati Docker online: https://riptutorial.com/it/docker/topic/1318/volumi-dati-docker

Capitolo 37: Volumi di dati e contenitori di dati

Examples

Contenitori di soli dati

I contenitori di soli dati sono obsoleti e ora sono considerati un anti-modello!

Nei giorni precedenti, prima del sottocomando del `volume` di Docker e prima che fosse possibile creare volumi con nome, Docker cancellava i volumi quando non vi erano più riferimenti a essi in alcun contenitore. I contenitori di soli dati sono obsoleti perché Docker ora offre la possibilità di creare volumi con nome, oltre a una maggiore utilità tramite il sottocomando dei vari `docker volume` docker. I contenitori di soli dati sono ora considerati un anti-pattern per questo motivo.

Molte risorse sul web degli ultimi due anni menzionano l'utilizzo di un modello chiamato "contenitore solo dati", che è semplicemente un contenitore Docker che esiste solo per mantenere un riferimento a un volume di dati.

Ricorda che in questo contesto, un "volume di dati" è un volume Docker che non è montato dall'host. Per chiarire, un "volume di dati" è un volume che viene creato con la direttiva `VOLUME` Dockerfile o utilizzando l'opzione `-v` sulla riga di comando in un comando di `docker run` , in particolare con il formato `-v /path/on/container` . Pertanto un "contenitore di soli dati" è un contenitore il cui unico scopo è quello di avere un volume di dati collegato, che viene utilizzato `--volumes-from` flag `--volumes-from` in un comando di `docker run` . Per esempio:

```
docker run -d --name "mysql-data" -v "/var/lib/mysql" alpine /bin/true
```

Quando viene eseguito il comando precedente, viene creato un "contenitore solo dati". È semplicemente un contenitore vuoto con un volume di dati collegato. È stato quindi possibile utilizzare questo volume in un altro contenitore in questo modo:

```
docker run -d --name="mysql" --volumes-from="mysql-data" mysql
```

Il contenitore `mysql` ora ha lo stesso volume che è anche in `mysql-data` .

Poiché Docker ora fornisce il sottocomando del `volume` e i volumi con nome, questo modello è ora obsoleto e non consigliato.

Per iniziare con il sottocomando del `volume` e i volumi con nome vedere Creazione di un volume denominato

Creazione di un volume di dati

```
docker run -d --name "mysql-1" -v "/var/lib/mysql" mysql
```

Questo comando crea un nuovo contenitore dall'immagine `mysql` . Crea anche un nuovo volume di dati, che poi monta nel contenitore in `/var/lib/mysql` . Questo volume aiuta tutti i dati al suo interno a persistere oltre la durata del contenitore. Vale a dire, quando un contenitore viene rimosso, vengono rimosse anche le sue modifiche al filesystem. Se un database stava memorizzando i dati nel contenitore e il contenitore è stato rimosso, anche tutti i dati vengono rimossi. I volumi rimarranno in una posizione particolare anche oltre il momento in cui viene rimosso il contenitore.

È possibile utilizzare lo stesso volume in più contenitori con l'opzione della riga di comando `--volumes-from` :

```
docker run -d --name="mysql-2" --volumes-from="mysql-1" mysql
```

Il contenitore `mysql-2` ora ha il volume di dati di `mysql-1` ad esso collegato, usando anche il percorso `/var/lib/mysql` .

Leggi Volumi di dati e contenitori di dati online: https://riptutorial.com/it/docker/topic/3224/volumi-di-dati-e-contenitori-di-dati

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Docker	abaracedo, Aminadav, Braiam, Carlos Rafael Ramirez, Community, ganesshkumar, HankCa, Josha Inglis, L0j1k, mohan08p, Nathaniel Ford, schumacherj, Siddharth Srinivasan, SztupY, Vishrant
2	API del motore Docker	Ashish Bista, atv, BMitch, L0j1k, Radoslav Stoyanov, SztupY
3	Checkpoint e ripristino dei contenitori	Bastian, Fuzzyma
4	Collegamento di contenitori	Jett Jones
5	Come configurare la replica Mongo a tre nodi utilizzando l'immagine di Docker e Provisioned utilizzando Chef	Innocent Anigbo
6	Come eseguire il debug quando la creazione della finestra mobile non riesce	user2915097
7	Concetto di volumi Docker	Amit Poonia, Rob Bednark, serieznyi
8	Costruire immagini	cjsimon, ETL, Ken Cochrane, L0j1k, Nathan Arthur, Nathaniel Ford, Nour Chawich, SztupY, user2915097, Wolfgang
9	Creare un servizio con persistenza	Carlos Rafael Ramirez, Vanuan
10	Debug di un contenitore	allprog, Binary Nerd, foraidt, L0j1k, Nathaniel Ford, user2915097, yadutaf
11	Docker in Docker	Ohmen

12	Docker Machine	Amine24h, kubanczyk, Nik Rahmel, user2915097, yadutaf
13	Docker network	HankCa, L0j1k, Nathaniel Ford
14	Dockerfiles	BMitch, foraidt, k0pernikus, kubanczyk, L0j1k, ob1, Ohmen, rosysnake, satsumas, Stephen Leppik, Thiago Almeida, Wassim Dhif, yadutaf
15	Esecuzione di contenitori	abaracedo, Adri C.S., AlcaDotS, atv, Binary Nerd, BMitch, Camilo Silva, Carlos Rafael Ramirez, cizixs, cjsimon, Claudiu, ElMesa, Emil Burzo, enderland, Felipe Plets, ganesshkumar, Gergely Fehérvári, ISanych, L0j1k, Nathan Arthur, Patrick Auld, RoyB, ssice, SztupY, Thomasleveil, tommyyards, VanagaS, Wolfgang, zinking
16	Esecuzione di semplice applicazione Node.js	Siddharth Srinivasan
17	eseguire console in docker 1.12 sciame	Jilles van Gurp
18	Eventi Docker	Nathaniel Ford, user2915097
19	finestra mobile ispeziona i vari campi per la chiave: valore ed elementi della lista	user2915097
20	Gestione dei contenitori	akhyar, atv, Binary Nerd, BrunoLM, Carlos Rafael Ramirez, Emil Burzo, Felipe Plets, ganesshkumar, L0j1k, Matt, Nathaniel Ford, Rafal Wiliński, Sachin Malhotra, serieznyi, sk8terboi87 ツ, tommyyards, user2915097, Victor Oliveira Antonino, Wolfgang, Xavier Nicollet, zygimantus
21	Gestire le immagini	akhyar, Björn Enochsson, dsw88, L0j1k, Nathan Arthur, Nathaniel Ford, Szymon Biliński, user2915097, Wolfgang, zygimantus
22	Iptables con Docker	Adrien Ferrand
23	Ispezionando un contenitore funzionante	AlcaDotS, devopskata, Felipe Plets, h3nrik, Jilles van Gurp, L0j1k, Milind Chawre, Nik Rahmel, Stephen Leppik, user2915097, yadutaf
24	Limitazione dell'accesso alla rete del contenitore	xeor

25	Modalità Docker --net (bridge, hots, contenitore mappato e nessuno).	mohan08p
26	Modalità sciame Docker	abronan, Christian, Farhad Farahi, Jilles van Gurp, kstromeiraos, kubanczyk, ob1, Philip, Vanuan
27	Ordinamento contenuti Dockerfile	akhyar, Philip
28	passaggio di dati segreti a un contenitore in esecuzione	user2915097
29	Più processi in un'istanza contenitore	h3nrik, Ohmen, Xavier Nicollet
30	Registrazione	Jilles van Gurp, Vanuan
31	Registro di sistema privato / sicuro Docker con API v2	bastien enjalbert, kubanczyk
32	Registro Docker	Ashish Bista, L0j1k
33	Servizi in corso	Mateusz Mrozewski, Philip
34	sicurezza	user2915097
35	Statistiche Docker tutti i contenitori in esecuzione	Kostiantyn Rybnikov
36	Volumi dati Docker	James Hewitt, L0j1k, NRKirby, Nuno Curado, Scott Coates, t3h2mas
37	Volumi di dati e contenitori di dati	GameScripting, L0j1k, melihovv