はじめに

Docker Engine ユーザガイド〜リファレンス編

このガイドについて

Docker Engine ドキュメント（https://docs.docker.com/）の日本語翻訳版（http://docs.docker.jp/）をもとに電子データ化しました。

免責事項

現時点ではベータ版であり、内容に関しては無保証です。PDF の評価用として公開しました。Docker は活発な開発と改善が続いています。同様に、ドキュメントもオリジナルですら日々書き換えが進んでいますので、あらかじめご了承願います。

このドキュメントに関する問題や翻訳に対するご意見がありましたら、GitHub リポジトリ上の Issue までご連絡いただくか、pull request をお願いいたします。

- https://github.com/zembutsu/docs.docker.jp

履歴

- 2016 年 5 月 20 日　リファレンス編 beta1 を公開

目次

Appendix.1

Docker 用語集

A1.1 用語集

Docker プロジェクト界隈で使われる用語の一覧です。

aufs（エーユーエフエス）

aufs（advanced multi layered unification filesystem；複数のレイヤを統合した高度なファイルシステム、の意味）は、Docker がストレージ用のバックエンドとしてサポートする Linux のファイルシステムです。

ベース・イメージ（base image）

親イメージを持たないイメージを、 ベース・イメージと呼びます。

boot2docker

boot2docker[*1]（ブート・トゥ・ドッカー）は Docker コンテナの実行に特化した Linux ディストリビューションです。Mac および Windows 向けの boot2docker は、Docker Toolbox のインストールに含まれる `docker-machine` に置き換えられました。

btrfs

btrfs（B-tree file system；ビー・ツリー・ファイルシステム、バター・エフエス）は、Docker がストレージ用のバックエンドとしてサポートする Linux のファイルシステムです。

ビルド（build）

ビルド（build）とは、`Dockerfile` を使って Docker イメージを構築する方法です。構築時には Dockerfile と「コンテクスト」（内容物の意味）を使います。コンテクストとは、イメージ構築に必要なファイル群が置かれているディレクトリです。

*1 http://boot2docker.io/

cgroups

cgroups（control groups；コントロール・グループ）は Linux カーネルの機能であり、プロセスの集合が使うリソース（CPU、メモリ、ディスク I/O、ネットワーク等）を制限・計算・隔離します。Docker はリソース上限の管理と隔離に cgroups を使います。

cgroups の別名：control groups

Compose

Compose（コンポーズ）は、Docker で複雑なアプリケーションの実行と定義をするツールです。Compose を使えば、1 つのファイルに複数のコンテナ・アプリケーションを定義しておき、コマンドを 1 つ実行するだけで、アプリケーションを使うために必要な全てを実行します。

Compose の別名： docker-compose、fig

コンテナ

コンテナ（container）は docker イメージを実行する時の実体（runtime instance）です。

Docker コンテナには、次のものを含みます。

- Docker イメージ
- 実行環境
- 命令の標準セット

Docker コンテナの概念は、輸送用のコンテナから拝借したものです。コンテナは「物」を世界的に輸送するために標準が定義されています。Docker はソフトウェアを送るための標準を定義しています。

データ・ボリューム

データ・ボリューム（data volume）は、コンテナ内部でユニオン・ファイル・システムを迂回するため特別に設計されたディレクトリです。データ・ボリュームは長期的なデータ保管のために設計されており、コンテナのライフサイクルからは独立しています。そのため、コンテナを削除してもボリュームが自動的に消されることは有り得ませんし、コンテナから参照されなくなったボリュームが「掃除」（garbage collect）されることもありません。

Docker

Docker（ドッカー）には次の意味があります。

- Docker プロジェクト全体を指す言葉であり、開発者やシステム管理者がアプリケーションを開発・移動・実行するためのプラットフォームです。
- ホスト上で動く docker デーモンプロセスであり、イメージとコンテナを管理します。

Docker Hub

Docker Hub（ドッカー・ハブ）は Docker とこのコンポーネントで動くリソースを集めた場所です。以下のサービスを提供します。

- Docker イメージを預かる（ホスティング）
- ユーザ認証
- イメージの自動構築と、構築トリガ（build triggers）やウェブ・フック（web hooks）のようなワークフロー・ツール
- GitHub および Bitbucket との統合

Dockerfile

Dockerfile（ドッカーファイル）はテキスト形式のドキュメントであり、通常 Docker イメージを構築するために手動で実行が必要になる全てのコマンドを含みます。Docker は Dockerfile の命令を読み込み、自動的にイメージを構築します。

ファイルシステム

ファイルシステムとは、オペレーティング・システムがファイルに名前を付け、かつ、効率的な保管と修正のためにファイルに場所を割り当てます。

例：

- Linux : ext4, aufs, btrfs, zfs
- Windows : NTFS
- OS X : HFS+

イメージ

Docker イメージはコンテナの元です。イメージとはルート・ファイルシステムに対する変更を並べ集めたもので、コンテナを実行する間に使われる実行パラメータに相当します。典型的なイメージはユニオン・ファイル・システムの層（スタック）がお互いに積み重なっています。イメージは状態を保持せず、変更もできません。

libcontainer

libcontainer（リブコンテナ）は Go 言語のネイティブな実装であり、名前空間・cgroup・機能・ファイルシステムへのアクセス管理を持つコンテナを作成します。コンテナを作成後、コンテナに対してライフサイクル上の追加操作を可能にします。

libnetwork

libnetwork（リブネットワーク）は Go 言語のネイティブな実装であり、コンテナのネットワーク名前空間や他のネットワーク・リソースを作成・管理します。コンテナを作成後、コンテナに対してライフサイクル上の追加操作を可能にします。

リンク機能（link）

リンク機能は同じホスト上で実行している Docker コンテナ間を接続するための、レガシーな（古い）インター

フェースです。リンク機能を使うと、ホスト側のネットワーク・ポートを開く必要がありません。現在は、この機能の替わりに Docker ネットワーク機能を使います。

Machine

Machine（マシン）は Docker ホストを簡単に作成できるようにするツールであり、クラウド・プロバイダ上やデータセンタでも利用できます。Machine はサーバを作成し、そこに Docker をインストールし、Docker クライアントで通信できるように設定します。

別名： docker-machine

オーバレイ・ネットワーク・ドライバ

オーバレイ・ネットワーク・ドライバ（overlay network driver）は、クラスタ上の Docker コンテナに対して、複数ホスト間のネットワーク接続性を簡単に提供します。

オーバレイ・ストレージ・ドライバ

OverlayFS（オーバレイエフエス）は、他のファイルシステムに対するユニオン・マウントを Linux に実装するもので、ファイルシステム向けのサービスです。

レジストリ（registry）

レジストリ（registry）とはイメージを持つリポジトリを預かるサービス（ホステッド・サービス）であり、レジストリ API に応答します。

デフォルトのレジストリにアクセスするには、ブラウザで Docker Hub を開くか、`docker search` コマンドを使います。

リポジトリ

リポジトリ（repository）とは Docker イメージの集まりです。リポジトリはレジストリ・サーバに送信すると、共有されるようにできます。リポジトリの中では、イメージの違いをタグでラベル付けします。

Swarm

Swarm（スウォーム）は Docker 用のネイティブなクラスタリング・ツールです。Swarm は複数の Docker ホストを一緒にまとめ（プールする）、1 つの仮想的な Docker ホストのように装います。Swarm は標準 Docker API を提供するため、既に Docker で使えるツールであれば、複数のホスト上で透過的にスケールさせることができます。

別名：docker-swarm

タグ

タグ（tag）は リポジトリ上の Docker イメージに割り当てるラベルです。タグを使い、リポジトリ上のイメージを互いに識別します。

ここでのラベルとは、docker デーモン用のキー・バリューで設定するラベルとは関係がありません。

Toolbox

Docker Toolbox（ツールボックス）は Mac あるいは Windows ユーザ向けのインストーラです。

ユニオン・ファイル・システム

ユニオン・ファイル・システム（Union file system）や UnionFS（ユニオンエフエス）は、非常に軽量で高速なレイヤを作成できるファイルシステムです。Docker はコンテナのブロック構築にユニオン・ファイル・システムを使います。

仮想マシン

仮想マシン（Virtual Machine）とは、コンピュータと疑似専用ハードウェアの全体をエミュレートするプログラムです。他のユーザと物理ハードウェアのリソースを共有しますが、オペレーティング・システムからは隔離されています。エンドユーザは専用ハードウェアと同じように仮想マシンを操作できます。

コンテナと比べると、仮想マシンの実行は重たいものですが、更なる隔離を提供し、自身でリソースを持っており、共有は最低限です。

別名：VM

Appendix.2

Docker コマンドライン・リファレンス

A2.1 docker run リファレンス

Docker は隔離（独立）したコンテナでプロセスを実行します。コンテナはホスト上で動くプロセスです。ホストとはローカルまたはリモートの環境です。作業者が docker run を実行したら、コンテナのプロセスを隔離（独立）して実行します。コンテナは自身ファイルシステムとネットワークを持ち、ホスト上の他のプロセス・ツリーからは隔離されて（独立して）います。

このセクションではコンテナ実行時にリソースを指定できるよう、docker run コマンドの使い方の詳細を説明します。

一般的な形式

基本的な docker run コマンドの形式は、次の通りです。

```
$ docker run [オプション] イメージ[:タグ|@ダイジェスト値] [コマンド] [引数...]
```

docker run コマンドは、コンテナ形成の元になるイメージ指定が必要です。イメージの開発者はイメージに関連するデフォルト値を定義できます。

- デタッチドあるいはフォアグラウンドで実行
- コンテナの識別
- ネットワーク設定
- 実行時の CPU とメモリの制限
- 権限と LXC 設定

docker run [オプション] の実行時、作業者は設定の追加や、開発者がイメージに指定したデフォルト設定を上書き可能です。そして更に、作業者は Docker の実行時、ほぼ全てのデフォルト設定を上書きできます。run コマンドは他の docker コマンドより多くのオプションがあるため、作業者はイメージと Docker 実行時のデフォルト設定を上書きできます。

様々な種類の［オプション］を理解するには、「オプションの種類」のセクションをご覧ください。

Docker システムの設定によっては、docker run コマンドを sudo で実行する必要があるかもしれません。docker コマンドで sudo を使わないようにするには、システム管理者に docker という名称のグループの作成と、そこにユーザの追加を依頼してください。この設定に関するより詳しい情報は、各オペレーティング・システム向けのインストール用ドキュメントをご覧ください。

作業者専用のオプション

作業者（ docker run の実行者 ）のみ、以下のオプションを設定できます。

- デタッチドまたはフォアグラウンド
- コンテナの識別
- IPC 設定
- ネットワーク設定
- 再起動ポリシー
- クリーンアップ
- 実行時のリソース制限
- 実行時の権限、Linux 機能、LXC 設定

デタッチドまたはフォアグラウンド

Docker コンテナの起動時に、まず、コンテナをバックグラウンドで**「デタッチド」モード（detached mode）**で実行するか、デフォルトの**フォアグラウンド・モード（foreground mode）**で実行するかを決める必要があります。

-d=false: デタッチド・モード：コンテナをバックグラウンドで実行し、新しいコンテナ ID を表示

デタッチド(-d)

コンテナをデタッチド・モードで起動するには、 -d=true か -d オプションを使います。設計上、コンテナが実行するルート・プロセスが終了したら、デタッチド・モードで起動したコンテナも終了します。デタッチド・モードのコンテナは停止しても自動的に削除できません。つまり -d オプションでは --rm を指定できません。

デタッチドのコンテナでは service x start コマンドを受け付けません。例えば、次のコマンドは nginx サービスの起動を試みます。

```
$ docker run -d -p 80:80 my_image service nginx start
```

コンテナ内で nginx サービスの起動は成功します。しかしながら、デタッチド・コンテナの枠組み内では処理に失敗します。これはルート・プロセス（service nginx start）が終了するため、デタッチド・コンテナは停止されます。その結果、nginx サービスは実行しますが、実行を継続できません。この方法を使わず nginx ウェブ・サーバのプロセスを実行するには、次のようにします。

```
$ docker run -d -p 80:80 my_image nginx -g 'daemon off;'
```

コンテナの入出力はネットワーク接続や共有ボリュームも扱えます。コマンドラインで docker run を実行し終わった後でも、必要になる場合があるでしょう。

デタッチド・コンテナに再度アタッチ（接続）するには、docker attach コマンドを使います。

フォアグラウンド

フォアグラウンド・モード（ -d を指定しないデフォルト）の場合、docker run はコンテナの中でプロセスを開始し、プロセスの標準入出力・標準エラーをコンソールにアタッチします。これは TTY の振りをするだけでなく

（TTY は大部分のコマンド・ラインで実行可能なものと想定しています）、シグナルも渡せます。

```
-a=[]           : 「標準入力」「標準出力」かつ／または「標準エラー出力」にアタッチ
-t=false        : 疑似ターミナル（pseudo-tty）の割り当て
--sig-proxy=true: 受信したシグナルをプロセスに中継（ターミナルモード以外のみ）
-i=false        : アタッチしていなくても、標準入力をオープンに維持
```

Docker で -a を指定しなければ、Docker は自動的に全ての標準ストリームをアタッチします。３つの標準ストリーム（STDIN、STDOUT、STDERR）のうち、特定のものに対してのみ接続も可能です。

```
$ docker run -a stdin -a stdout -i -t ubuntu /bin/bash
```

（シェルのような）インタラクティブなプロセスでは、コンテナのプロセスに対して tty を割り当てるために、 -i -t を一緒に使う必要があります。 -i -t は -it と書けます。後の例で出てきます。 -t を指定したら、クライアント側の出力を echo test | docker run -i busybox cat のようにリダイレクトやパイプできます。

コンテナ内で PID 1 として実行しているプロセスは、Linux が特別に扱います。デフォルトの操作では、あらゆるシグナルを無視します。そのため、プロセスは SIGINT か SIGTERM で停止するようにコードを書かない限り、停止できません。

コンテナの識別

名前（--name）

作業者はコンテナを３つの方法で識別できます。

- 長い（ロング）UUID 識別子（" f78375b1c487e03c9438c729345e54db9d20cfa2ac1fc3494b6eb60872e74778 "）
- 短い UUID（ショート）識別子（" f78375b1c487 "）
- 名前（" evil_ptolemy "）

UUID 識別子は Docker デーモンから与えられます。コンテナの名前を --name オプションで割り当てなければ、デーモンはランダムな文字列から名前を生成します。コンテナに対する目的を表すためには、name の定義が簡単な方法でしょう。name を指定したら、これを Docker ネットワーク内でコンテナを参照用に使えます。この参照機能は、バックグラウンドでもフォアグラウンドどちらの Docker コンテナでも動作します。

デフォルトのブリッジ・ネットワーク内にあるコンテナの場合は、相互に名前で通信するにはリンクする必要があります。

PID 相当の機能

あとは、自動処理を簡単にするため、Docker は任意に選択したファイルに対してコンテナ ID を書き出せます。これは、プログラムがプロセス ID をファイルに書き出す（いわゆる PID ファイルです）のに似ています。

```
--cidfile="": コンテナの ID をファイルに書き出す
```

イメージ[:タグ]

コンテナ実行時のコマンドで**イメージ[:タグ]**を追加すると、イメージのバージョンを厳密に指定できます。例えば docker run ubuntu:14.04 と実行します。

イメージ[@ダイジェスト値]

イメージ形式 v2 以降のイメージを使えば、その中に**ダイジェスト値（digest）**と呼ばれる識別子が、内容に対して割り当てられています。入力に使われたイメージファイルに対する変更が無ければ、ダイジェスト値とは予想されうる値であり、参照可能なものです。

PID 設定（--pid）

```
--pid="" ：コンテナに対する PID（プロセス）名前空間モードを指定
      'host':コンテナ内のホストが使う PID 名前空間
```

デフォルトでは、全てのコンテナは有効な **PID 名前空間**を持っています。

PID 名前空間はプロセスの分離をもたらします。PID 名前空間はシステム・プロセスを見えないようにし、PID 1 を含むプロセス ID を再利用できるようにします。

コンテナがホスト上の特定のプロセス名前空間を共有する場合は、コンテナ内のプロセスが、システム上の全プロセスを基本的に見られるようにします。例えば、strace や gdb のようなデバッグ用ツールを含むコンテナを構築した時、コンテナ内のデバッグ用プロセスのみツールを使えるように指定する場合です。

```
$ docker run --pid=host rhel7 strace -p 1234
```

これはホスト上の pid 1234 にあるコンテナ内で strace を使うコマンドです。

UTS 設定（--uts）

```
--uts="" ：UTS 名前空間モードをコンテナに設定する
      'host': コンテナ内でホストの UTS 名前空間を使用
```

UTS 名前空間とは、プロセスを実行する名前空間上で見えるホスト名とドメイン名を設定するものです。デフォルトでは、全てのコンテナは --uts=host の指定により、自身の UTS 名前空間を持っています。host には、ホスト名として同じ UTS 名前空間をコンテナで使えるよう設定します。なお、host UTS モードでは --hostname の指定ができないため、ご注意ください。

ホスト上と UTS 名前空間を共有したい場合もあるでしょう。例えば、コンテナを動かすホストがホスト名を変更してしまい、コンテナのホスト名も変更したい場合です。より高度な使い方としては、コンテナからホスト側のホスト名の変更を行うケースです。

IPC 設定（--ipc）

```
--ipc="" ：コンテナに IPC モードを設定する
            'container:<名前|id>': 他のコンテナの IPC 名前空間を再利用
            'host': ホストの IPC 名前空間をコンテナの中で使用
```

デフォルトでは、全てのコンテナが有効な **IPC 名前空間**を持っています。

IPC（POSIX/SysV IPC）名前空間は、共有メモリ・セグメント、セマフォ、メッセージ・キューと呼ばれる分離を提供します。

プロセス間通信は共有メモリ・セグメントはメモリの速度まで（ネットワーク・スタックをパイプするか通過するよりも速く加速）します。共有メモリとは、一般的にデータベースや、科学計算や緊急サービス産業向けの高性能アプリケーション向けカスタム・ビルド（典型的なのは、C/OpenMPI、C++ の高速化ライブラリ）に用いられます。この種のアプリケーションが複数のコンテナに分割される場合は、コンテナの IPC 機構を使って共有する必要があるでしょう。

ネットワーク設定

```
--dns=[]         : コンテナ用のカスタム DNS サーバを設定
--net="bridge"   : コンテナをネットワークに接続
                   'bridge': docker ブリッジ上でコンテナ用に新しいネットワーク・スタックを作成
                   'none': コンテナにネットワーク機能を付けない
                   'container:<name|id>': 他のコンテナ用ネットワーク・スタックを再利用
                   'host': コンテナ内でホスト側ネットワーク・スタックを使用
                   'NETWORK': 「docker network create」コマンドでユーザ作成したネットワークに作成
--net-alias=[]   : コンテナにネットワーク内部用のエイリアスを追加
--add-host=""    : /etc/hosts に行を追加（ホスト名:IP アドレス）
--mac-address="" : コンテナのイーサネット・デバイス Mac アドレスを指定
--ip=""          : コンテナのイーサネット・デバイスに IPv4 アドレスを指定
--ip6=""         : コンテナのイーサネット・デバイスに IPv6 アドレスを指定
```

デフォルトでは、全てのコンテナはネットワーク機能を持っており、外部に対する接続が可能です。作業者がネットワークを無効化したい場合は `docker run --net=none` を指定し、内側と外側の両方のネットワーク機能を無効化します。このように指定したら、I/O 処理はファイルに対してか、STDIN と STDOUT のみになります。

公開用のポートを他のコンテナとリンクできるのは、デフォルト（ブリッジ）のみです。リンク機能はレガシー（過去の）機能です。リンク機能を使うよりも、常に Docker ネットワーク機能を使うべきです。

コンテナは、デフォルトではホストと同じ DNS サーバを使いますが、 `--dns` で上書きできます。

デフォルトでは、コンテナに割り当てられる IP アドレスを使い、MAC アドレスを生成します。コンテナの MAC アドレスの指定は、 `--mac-address` パラメータ（書式：`12:34:56:78:9a:bc`）を使い MAC アドレスを指定できます。Docker は MAC アドレスがユニークかどうか（重複しているかどうか）を確認する仕組みが無いため、ご注意ください。

サポートしているネットワーク：

ネットワーク	説明
none	コンテナにネットワーク機能を持たせません。
bridge（デフォルト）	コンテナを各インターフェースに接続します。
host	コンテナ内でホスト側のネットワーク・スタックを使います。
container:<名前|id>	他のコンテナ名か ID を指定し、そのネットワーク・スタックを使います。
ユーザ定義ネットワーク	ユーザが作成したネットワーク（`docker network create` コマンドを使用）にコンテナを接続します。

ネットワーク：none

コンテナのネットワークを `none` に指定したら、外部の経路に対してアクセス不能になります。コンテナ内では loopback （ループバック）インターフェースが有効ですが、外部のトラフィックに対する経路が無くなります。

ネットワーク：bridge

コンテナのネットワークを bridge に指定したら、コンテナは Docker のデフォルト・ネットワーク機能をセットアップします。ブリッジはホスト上で設定されるもので、通常は docker0 と名前が付けられます。そして、veth インターフェースのペアがコンテナ用に作成されます。veth ペアの片方はホスト側にアタッチされたままとなります。もう一方は、コンテナの名前空間の中で loopback インターフェースに加えて追加します。ブリッジ・ネットワーク上で IP アドレスがコンテナに割り当てられ、コンテナに対するトラフィックはこのブリッジを経由します。

デフォルトでは、コンテナは各々の IP アドレスを経由して通信できます。コンテナ名で通信するには、リンクする必要があります。

ネットワーク：host

host ネットワークをコンテナに設定したら、ホスト側のネットワーク・スタックと、全てのホスト上のインターフェースがコンテナ上でも共有できます。コンテナのホスト名はホストシステム上のホスト名と一致します。host ネットワーク・モードでは、 --add-host、 --hostname、 --dns、 --dns-search、 --dns-opt、 --mac-address が無効になるのでご注意ください。 たとえ host ネットワーク・モードだとしても、コンテナは自身の UTS 名前空間をデフォルトで持ちます。そのため、host ネットワーク・モードで --hostname が許可されるのは、コンテナの中でホスト名を変えるだけです。

デフォルトの bridge モードと比べ、host モードは著しくネットワーク性能が優れます。これは、bridge の場合は docker デーモンの仮想化レベルを通過しているのに対して、host の場合はネイティブなネットワーク・スタックを用いるからです。例えば、プロダクションのロードバランサや高性能のウェブサーバのような、ネットワーク性能がクリティカルな環境では、このモードでのコンテナ動作を推奨します。

--net="host"の指定時は、コンテナは D-bus のようなローカル・システム・サービスに対してフルアクセス可能なため、安全ではないと考えられます。

ネットワーク：container

container ネットワークをコンテナに指定したら、他のコンテナのネットワーク・スタックを共有します。他のコンテナ名は --net container:<名前|id> の書式で指定する必要があります。container ネットワーク・モードでは、 --add-host、 --hostname、 --dns、 --dns-search、 --dns-opt、 --mac-address が無効になるだけでなく、 --publish、 --publish-all、 --expose も無効になるのでご注意ください。

次の例は、Redis コンテナで Redis が localhost をバインドしている時、localhost インターフェースを通して Redis サーバに redis-cli コマンドを実行して接続します。

```
$ docker run -d --name redis example/redis --bind 127.0.0.1
$ # redis コンテナのネットワーク・スタックにある localhost にアクセスします
$ docker run --rm -it --net container:redis example/redis-cli -h 127.0.0.1
```

ユーザ定義ネットワーク

ネットワークを作成するには、Docker ネットワーク・ドライバか外部のネットワーク・ドライバ・プラグインを使います。同じネットワークに対して、複数のコンテナが接続できます。ユーザ定義ネットワークに接続したら、コンテナはコンテナの名前や IP アドレスを使い、簡単に通信できるようになります。

overlay ネットワークやカスタム・プラグインは、複数のホストへの接続性をサポートしています。コンテナが同一のマルチホスト・ネットワークに接続していれば、別々のエンジンで起動していても、このネットワークを通して通信可能です。

以下の例は、内部 bridge ネットワーク・ドライバを使ってネットワークを作成し、作成したネットワーク上でコンテナを実行します。

```
$ docker network create -d bridge my-net
$ docker run --net=my-net -itd --name=container3 busybox
```

/etc/hosts の管理

/etc/hosts には localhost や一般的な項目と同じように、自分が定義したコンテナのホスト名を追加できます。/etc/hosts に行を追加するには --add-host フラグを使います。

```
$ docker run -it --add-host db-static:86.75.30.9 ubuntu cat /etc/hosts
172.17.0.22     09d03f76bf2c
fe00::0         ip6-localnet
ff00::0         ip6-mcastprefix
ff02::1         ip6-allnodes
ff02::2         ip6-allrouters
127.0.0.1       localhost
::1             localhost ip6-localhost ip6-loopback
86.75.30.9      db-static
```

コンテナがデフォルト・ブリッジ・ネットワークに接続し、他のコンテナと link（リンク）すると、コンテナの /etc/hosts ファイルが更新され、リンクされたコンテナ名が書き込まれます。

もしもコンテナがユーザ定義ネットワークに接続した場合は、コンテナの /etc/hosts ファイルが更新され、ユーザ定義ネットワーク上の他のコンテナ名が書き込まれます。

Docker がコンテナの /etc/hosts ファイルをリアルタイムに更新するかもしれません。そのため、コンテナ内のプロセスが /etc/hosts ファイルを読み込もうとしても空だったり、あるいは最後まで読み込めなかったりする場合が有り得ます。ほとんどの場合、再読み込みで問題が解決するでしょう。

再起動ポリシー（--restart）

Docker で実行時に --restart フラグを使えば、再起動ポリシーを指定できます。再起動ポリシーとは、コンテナが終了時に再起動すべきかどうかを定義します。

コンテナの再起動ポリシーが有効な場合、 docker ps でコンテナを見たら、常に Up か Restarting のどちらかです。また、再起動ポリシーが有効かどうかを確認するため、docker events を使うのも便利です。

Docker は以下の再起動ポリシーをサポートしています。

ポリシー	説明
no（なし）	コンテナが終了しても、自動的には再起動しません。これがデフォルトです。
on-failure[:最大リトライ数]	コンテナが 0 以外の終了コードで停止したら再起動します。オプションで Docker デーモンが何度再起動を試みるかを指定できます。
always（常に）	コンテナの終了コードに拘わらず、常にコンテナの再起動を試みます。Docker デーモンは無制限に再起動を試みます。また、デーモンの起動時にも、コンテナの状況に拘わらず常に起動を試みます。
unless-stopped（停止していない場合）	コンテナの終了コードに拘わらず、常にコンテナの再起動を試みます。しかし、直近のコンテナが停止状態であったのであれば、デーモンの起動時にコンテナを開始しません。

サーバが溢れかえるのを防ぐため、再起動の前に遅延時間が追加されます（遅延は 100 ミリ秒から開始し、直前の値の 2 倍になります)。つまり、デーモンは 100 ミリ秒待った後は、200 ミリ秒、400、800、1600 …と on-failure 上限に到達するか、あるいは、コンテナを docker stop で停止するか、 docker rm -f で強制削除するまで続けます。

コンテナの再起動が成功すると（コンテナは少なくとも 10 秒以内で起動します)、遅延時間の値は再び 100 ミリ秒にリセットされます。

on-failure ポリシーを使えば、Docker がコンテナの再起動を試みる最大回数を指定できます。デフォルトでは、Docker はコンテナを永久に再起動し続けます。コンテナの再起動（を試みる）回数は docker inspect で確認可能です。例えば、コンテナ「my-container」の再起動数を取得するには、次のようにします。

```
$ docker inspect -f "{{ .RestartCount }}" my-container
# 2
```

あるいは、コンテナが（再）起動した時刻を知るには、次のようにします。

```
$ docker inspect -f "{{ .State.StartedAt }}" my-container
# 2015-03-04T23:47:07.691840179Z
```

再起動ポリシーとクリーンアップは同時に指定できません。 --restart と --rm を同時に指定してもエラーになります。

例

```
$ docker run --restart=always redis
```

こちらの例は、 常に（always）再起動するポリシーで redis コンテナを実行します。そのため、コンテナが停止したら Docker はコンテナを再起動します。

```
$ docker run --restart=on-failure:10 redis
```

こちらの例は、 失敗したら（on-failure） 10 回カウントするまで再起動を行うポリシーで redis コンテナを起動しています。もし redis コンテナが 0 以外の状態で終了したら、Docker はコンテナの再起動を 10 回続けて試みます。再起動の上限を設定できるのは、on-failure ポリシーを有効にした場合のみです。

終了ステータス（exit status）

docker run の終了コードから得られる情報は、なぜコンテナが実行に失敗したかや、なぜ終了したかです。docker run がゼロ以外のコードで終了する時、終了コードは chroot 標準に従います。

125 は **Docker デーモン自身**のエラー発生です。

```
$ docker run --foo busybox; echo $?
# 定義されていない --foo フラグを指定したため
  See 'docker run --help'.
  125
```

126 は**コンテナ内のコマンド**が実行できない場合のエラーです。

```
$ docker run busybox /etc; echo $?
# "/etc" には実行権限がありません
  docker: Error response from daemon: Contained command could not be invoked
```

```
126
```

127 は**コンテナ内のコマンド**が見つからない場合です。

```
$ docker run busybox foo; echo $?
# 環境変数 $PATH の中に "foo" 実行ファイルが見つかりません。
  docker: Error response from daemon: Contained command not found or does not exist
  127
```

コンテナ内におけるコマンドの**終了コード**は上書きできます。

```
$ docker run busybox /bin/sh -c 'exit 3'
# 3
```

クリーンアップ（--rm）

デフォルトではコンテナを終了しても、コンテナのファイルシステム（の内容）を保持し続けます。これにより、多くのデバッグをより簡単にします（最後の状態を確認できるため）。そして、全てのデータを維持し続けるのがデフォルトです。しかし、短い期間だけフォアグラウンドで動かしたとしても、これらのコンテナのファイルシステムが溜まり続けます。そうではなく、 コンテナの終了時に、自動的にコンテナをクリーンアップし、ファイルシステムを削除する には --rm フラグを追加します。

```
--rm=false: 終了時に自動的にコンテテナを削除する（-d と同時に使用不可）
```

--rm フラグを設定したら、コンテナの削除時、関連するボリュームも削除します。これは docker rm -v my-container の実行と同じです。ただし、名前を指定しなかったボリュームのみ削除します。例えば docker run --rm -v /foo -v awesome:/bar busybox top の場合、 /foo ボリュームを削除します。しかし、 /bar は削除されません。 --volume-form で継承しているボリュームが削除されないのと同じ仕組みです。このように、オリジナルのボリュームに名前が指定されていれば、そこは削除**されません** 。

セキュリティ設定

```
--security-opt="label=user:USER"   : コンテナの user ラベルを指定
--security-opt="label=role:ROLE"   : コンテナの role ラベルを指定
--security-opt="label=type:TYPE"   : コンテナの type ラベルを指定
--security-opt="label=level:LEVEL" : コンテナの lovel ラベルを指定
--security-opt="label=disable"     : コンテナのラベル割り当てを無効化
--security-opt="apparmor=PROFILE"  : コンテナに適用する apparmor profile を指定
--security-opt="no-new-privileges" : コンテナが新しい権限を得るのを無効化
--security-opt="seccomp=unconfined": コンテナ用の seccomp 制限を無効化
--security-opt="seccomp=profile.json": sccomp フィルタで使うホワイトリスト syscall seccompo Json ファイルを指定
```

各コンテナに対するデフォルトのラベリング・スキーマ（labeling scheme）は --security-opt フラグを指定することで上書き可能です。例えば、MCS/MLS レベルを指定するには MLS システムが必要です。コンテナ間で同じ内容を共有できるようにレベルを指定するには、次のようにコマンドを実行します。

```
$ docker run --security-opt label=level:s0:c100,c200 -i -t fedora bash
```

MLS であれば、次のような例になります。

```
$ docker run --security-opt label=level:TopSecret -i -t rhel7 bash
```

コンテナに対するセキュリティ・ラベリングを無効化するには、 `--permissive` フラグを使い、次のように指定します。

```
$ docker run --security-opt label=disable -i -t fedora bash
```

コンテナ内のプロセスに対して、何らかのセキュリティ・ポリシーを適用するには、コンテナに対して何らかのタイプを指定します。コンテナを実行する時、Apache のポートのみがリッスンできるようにするには、次のように実行します。

```
$ docker run --security-opt label=type:svirt_apache_t -i -t centos bash
```

ここでは svirt_apache_t タイプに対する書き込みポリシーがあるものと想定しています。

コンテナのプロセスに特権を追加できないようにするには、次のコマンドを実行します。

```
$ docker run --security-opt no-new-privileges -it centos bash
```

より詳しい情報は、カーネルのドキュメント[*1] をご覧ください。

カスタム cgroups の指定

`--cgroup-parent` フラグを使うことで、コンテナを特定の cgroup で実行できるようにします。これにより自分自身で cgroup の作成や管理が可能になります。各 cgroup に対してカスタム・リソースを定義でき、コンテナを共通の親グループ下に置くこともできます。

実行時のリソース制限

作業者はコンテナのパフォーマンス・パラメータも調整できます。

オプション	説明
`-m , --memory=""`	メモリの上限（書式：<数値> [<単位>]、単位は b、k、m、g のいずれか）
`--memory-swap=""`	合計メモリの上限（メモリ＋スワップ、書式：<数値> [<単位>]、単位は b、k、m、g のいずれか）
`--memory-reservation=""`	メモリのソフト・リミット（書式：<数値> [<単位>]、単位は b、k、m、g のいずれか）
`--kernel-memory=""`	カーネル・メモリの上限（書式：<数値> [<単位>] 、単位は b、k、m、g のいずれか）
`-c , --cpu-shares=0`	CPU 共有（CPU shares）を相対値で指定
`--cpu-period=0`	CPU CFS(Completely Fair Scheduler) ピリオドの上限（訳者注：cgroup による CPU リソースへのアクセスを再割り当てする間隔）
`--cpuset-cpus=""`	実行する CPU の割り当て（0-3, 0,1）
`--cpuset-mems=""`	実行するメモリ・ノード（MEM）の割り当て（0-3, 0,1）。NUMA システムのみ

*1 https://www.kernel.org/doc/Documentation/prctl/no_new_privs.txt

	で動作
--cpu-quota=0	CPU CFS (Completely Fair Scheduler) のクォータを設定
--blkio-weight=0	ブロック I/O ウエイト（相対値）を 10 〜 1000 までの値でウエイトを設定
--oom-kill-disable=false	コンテナを OOM killer による停止を無効化するかどうか指定
--memory-swappiness=""	コンテナがメモリのスワップ度合いを調整。整数値の 0 〜 100 で指定

ユーザ・メモリの制限

ユーザのメモリ使用を制限するには、4 つの方法があります。

オプション	説明
memory=inf, memory-swap=inf（デフォルト）	コンテナに対する上限を設けない。コンテナは必要な分のメモリを使える
memory=L<inf, memory-swap=inf	（memory を指定し、memory-swap を -1 にする）コンテナは L バイト以上のメモリ使用が許されないが、必要があればスワップを使える（ホスト側がスワップ・メモリをサポートしている場合）
memory=L<inf, memory-swap=2*L	（memory を指定するが memory-swap は指定しない）コンテナは L バイト以上のメモリ使用は許されないが、指定した値の 2 倍の「追加」スワップ・メモリが使える
memory=L<inf, memory-swap=S<inf, L<=S	（memory も memory-swap も指定する）コンテナは L バイト以上のメモリ使用が許されないが、「追加」スワップ・メモリは S バイトまで使える

例：

```
$ docker run -ti ubuntu:14.04 /bin/bash
```

メモリを設定していません。これはコンテナ内のプロセスは必要な分だけメモリが使えます。それだけでなく、スワップ・メモリも同様に必要なだけ使えます。

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

メモリ上限を指定し、スワップ・メモリの制限を無効化しました。これはコンテナ内のプロセスは 300M のメモリを使えます。それだけでなく、スワップ・メモリは必要なだけ使えます（ホスト側がスワップ・メモリをサポートしている場合)。

```
$ docker run -ti -m 300M ubuntu:14.04 /bin/bash
```

メモリの上限のみ設定しました。これはコンテナが 300M のメモリと 300M のスワップ・メモリを使えます。合計の仮想メモリサイズ（total virtual memory size、 –memory-swap で指定）はメモリの 2 倍に設定されます。今回の例では、メモリ＋スワップは 2 × 300M ですので、プロセスは 300M のスワップ・メモリを利用できます。

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

メモリとスワップ・メモリを指定しましたので、コンテナ内のプロセスは 300M のメモリと 700M のスワップ・メモリを使えます。

メモリ予約（memory reservation）は、メモリに対するある種のソフト・リミットであり、共有メモリを大きくします。通常の状況下であれば、コンテナは必要とするだけ多くのメモリを使うことができます。そして、 -m か --memory オプションがある時のみ、コンテナに対してハード・リミットが設定されます。メモリ予約を設定したら、Docker はメモリのコンテンション（競合）や少ないメモリを検出し、コンテナが予約した上限まで使えるように

します。

メモリ予約の値は、常にハード・リミット以下に設定しなければ、ハード・リミットが先に処理されてしまいます。予約値を 0 に設定するのは、予約しないのと同じです。デフォルトでは（予約をセットしない場合)、メモリ予約とはメモリのハード・リミットと同じです。

メモリ予約とはソフト・リミット機能であり、制限を超過しないことを保証しません。その代わりに、かなりメモリが競合する場合、予約のヒント/設定に基づいてメモリの割り当てを試みる機能があります。

次の例はメモリの上限（-m）を 500M に制限し、メモリ予約を 200M に設定します。

```
$ docker run -ti -m 500M --memory-reservation 200M ubuntu:14.04 /bin/bash
```

この設定の下では、コンテナはメモリを 200MB 以上〜 500MB 以下まで使えます。次のシステム・メモリはコンテナのメモリが 200MB 以下になるよう縮小を試みます。

次の例はメモリのハード・リミットを設定せず、メモリ予約を 1G に設定します。

```
$ docker run -ti --memory-reservation 1G ubuntu:14.04 /bin/bash
```

コンテナはメモリを必要なだけ使えます。メモリ予約設定により、コンテナが長時間多くのメモリを消費しなくなります。これはコンテナがメモリを消費したとしても、予約分を使えるようにメモリの使用を縮小しようとするからです。

デフォルトでは、アウト・オブ・メモリ（OOM; out of memory）エラーが発生したら、カーネルはコンテナ内のプロセスを停止（kill）します。この振る舞いを変更するには、 --oom-kill-disable オプションを使います。また、 -m/--memory オプションを指定した時のみ、コンテナに対する OOM が無効化できます。もし -m フラグがセットされなければ、ホスト側でアウト・オブ・メモリ処理が発生します。また、ホスト側のシステム・プロセスが空きメモリを必要とするため、対象のプロセスを停止（kill）します。

次の例はメモリの上限を 100M とし、対象となるコンテナに対する OOM killer（アウト・オブ・メモリ処理による強制停止）を無効化します。

```
$ docker run -ti -m 100M --oom-kill-disable ubuntu:14.04 /bin/bash
```

次の例では、危険なフラグの使い方を説明します。

```
$ docker run -ti --oom-kill-disable ubuntu:14.04 /bin/bash
```

コンテナは無制限にメモリを使えるため、ホスト上のメモリを使い果たしたら、空きメモリ確保のために、システム・プロセスを停止する必要が出てきます。

カーネル・メモリ制限

カーネル・メモリはスワップ・アウトできないため、ユーザ・メモリとは根本的に異なります。このスワップができないことにより、システム・サービスがカーネル・メモリを多く使えないように妨害する可能性があります。カーネル・メモリとは、次の項目を指します。

- stack pages
- slab pages
- sockets memory pressure
- tcp memory pressure

これらのメモリを制限するため、カーネル・メモリの上限を設定できます。例えば、各プロセスが同じスタック

・ページ（stack page）を使うようにする場合です。カーネル・メモリの制限により、カーネル・メモリの使用量が大きい時、新しいプロセスの作成を妨げます。

カーネル・メモリはユーザ・メモリとは完全に独立しています。その代わり、ユーザ・メモリを制限すると同時に、カーネル・メモリの制限も必要です。上限の設定には3つの方法があります。ここでは、「U」はユーザ・メモリの上限で、「K」はカーネルの上限とみなしています。

オプション	結果
U != 0, K = inf （デフォルト）	カーネル・メモリが使う前に、標準的なメモリ制限を設ける仕組み。カーネル・メモリは完全に無視される。
U != 0, K < U	カーネル・メモリをユーザ・メモリのサブセットとする。この設定は cgroup ごとに大きな合計メモリ容量をオーバーコミットで割り当て、デプロイする場合に使いやすい。カーネル・メモリ制限のオーバーコミットは、全く推奨されていない。範囲が再利用できないメモリ領域の場合が有り得るため。この例では、K を設定したので、全グループの合計は、全メモリ容量を超えられない。そして、システム・サービスの品質のために U を任意に設定できる。
U != 0, K > U	カーネルのメモリを使用するため、コンテナ向けに両方のメモリが、ユーザ・カウンタと再利用トリガに影響を与えます。

例：

```
$ docker run -ti -m 500M – kernel-memory 50M ubuntu:14.04 /bin/bash
```

メモリとカーネルメモリを設定しました。これにより、コンテナ内のプロセスは合計 500M まで使えます。この 500M のメモリのうち、トップに 50M のカーネル・メモリがあります。

```
$ docker run -ti --kernel-memory 50M ubuntu:14.04 /bin/bash
```

-m オプションを指定せずカーネル・メモリを指定しました。そのため、コンテナ内のプロセスは必要なだけ多くのメモリを利用可能ですが、そこに最低限 50M のカーネル・メモリを使います。

スワップ回避（swappiness）制限

デフォルトでは、コンテナのカーネルは、アノニマス・ページ・メモリ上の何パーセントかをスワップ・アウトします。コンテナ向けのこのパーセントを指定するには --memory-swappiness で 0 〜 100 までの値を設定します。この値が 0 であればアノニマス・ページのスワッピング（anonymous page swapping）を無効にします。値を 100 にすると全てのページがスワップ可能となります。デフォルトでは、 --memory-swappiness を指定しなければ、メモリのスワップ回避（swappiness）は親の値を継承します。

例：

```
$ docker run -ti --memory-swappiness=0 ubuntu:14.04 /bin/bash
```

--memory-swappiness オプションが役立つのは、コンテナの作業セットを維持し、スワップによるパフォーマンスのペナルティを避ける場合です。

CPU 共有制限

デフォルトでは、全てのコンテナは同じ CPU サイクルの割合を持っています。この割合は変更可能なものであり、コンテナの CPU 共有ウエイトを、実行中の全てのコンテナに対する相対的な値として変更できます。

割合をデフォルトの 1024 から変更するには、 -c か --cpu-shares フラグでウエイトを 2 以上の値で設定します。もし 0 を設定しても、システムは値を無視してデフォルトの 1024 を使います。

割合が適用されるのは CPU に対する処理が集中する時のみです。あるコンテナのタスクがアイドル（何もしていない待機状態）であれば、他のコンテナは CPU 時間の余剰を利用できます。実際に割り当てられる CPU 時間の量は、システム上で実行するコンテナの下図に非常に依存します。

例えば、３つのコンテナがあるとしましょう。１つめの CPU 共有は 1024 で、残り２つの CPU 共有は 512 とします。もし３つのコンテナが CPU を 100%使用している状態になれば、１つめのコンテナが合計 CPU 時間の 50%を扱えます。４つめのコンテナを CPU 共有 1024 として追加したら、１つめのコンテナが得られるのは CPU の 33%になります。そして、残りの２つめ以降のコンテナが得られる CPU 時間は、それぞれ 16.5%（２つめ）、16.5%（３つめ）、33% （４つめ）となります。

複数のコアを持つ（マルチ・コア）システム上では、全ての CPU コアに分散し CPU 時間が共有されます。コンテナが CPU 時間の 100%より低く制限していても、個々の CPU コアでは 100%利用できます。

例えば、システムが３つ以上のコアを持っていると想定してみましょう。１つめのコンテナ{C0}では `-c=512` を指定し、１つのプロセスを実行するものとします。そして、他のコンテナ{C1}は `-c=1024` を指定し、２つのプロセスを実行するとします。この結果、CPU 共有は個々のコアに分散されます。

```
PID    container    CPU CPU share
100    {C0}     0   100% of CPU0
101    {C1}     1   100% of CPU1
102    {C1}     2   100% of CPU2
```

CPU 周期（period）制約

デフォルトの CPU CFS（Completely Fair Scheduler）周期は 100 ミリ秒です。コンテナの CPU 使用率を制限するには、 `--cpu-period` で CPU の周期を制限します。そして、通常の `--cpu-period` は `--cpu-quota` と一緒に使われるでしょう。

例：

```
$ docker run -ti --cpu-period=50000 --cpu-quota=25000 ubuntu:14.04 /bin/bash
```

もし１ CPU であれば、コンテナは 50 ミリ秒ごとに CPU の 50%を利用できます（訳者注：– cpu-quota のクォータ値が、 – cpu-period 周期の半分のため）。

より詳しい情報については、CFS ドキュメントの帯域制限について[*1] をご覧ください。

CPU セット制限

どの CPU でコンテナを実行するか指定できます。

例：

```
$ docker run -ti --cpuset-cpus="1,3" ubuntu:14.04 /bin/bash
```

これはコンテナ内のプロセスを cpu 1 と cpu 3 で実行します。

```
$ docker run -ti --cpuset-cpus="0-2" ubuntu:14.04 /bin/bash
```

こちらはコンテナ内のプロセスを cpu 0 、cpu 1 、 cpu 2 で実行します。

NUMA system 上でのみ、どのコンテナをメモリ上で実行するか設定できます。

*1 https://www.kernel.org/doc/Documentation/scheduler/sched-bwc.txt

```
$ docker run -ti --cpuset-mems="1,3" ubuntu:14.04 /bin/bash
```

この例ではコンテナ内でのプロセスを、メモリ・ノード 1 と 3 上のメモリのみに使用を制限します。

```
$ docker run -ti --cpuset-mems="0-2" ubuntu:14.04 /bin/bash
```

この例ではコンテナ内でのプロセスを、メモリ・ノード 0と1と2 上のメモリのみに使用を制限します。

CPU クォータ制限

--cpu-quota フラグはコンテナの CPU 使用を制限します。デフォルト値 0 の場合、コンテナは CPU リソース（1 CPU）の 100%を扱えます。CFS（Completely Fair Scheduler)がプロセス実行時のリソース割り当てを扱っており、これがカーネルによってデフォルトの Linux スケジューラとして使われています。この値を 50000 に指定したら、コンテナは CPU リソースの 50%までの使用に制限されます。複数の CPU の場合は、 --cpu-quota の調整が必要です。より詳しい情報については、CFS ドキュメントの帯域制限についてをご覧ください。

ブロック IO 帯域（blkio）制限

デフォルトでは、全てのコンテナはブロック IO 帯域（blkio）を同じ割合で取得します。デフォルトの割合は 500 です。割合を変更するには --blkio-weight フラグを使い、実行中の全てのコンテナに対する装置亜的な blkio ウエイトを指定します。

--blkio-weight フラグは、10 〜 1000 までのウエイト値を設定できます。例えば、次のコマンドは２つのコンテナに対し、別々の blkio ウエイトと設定しています。

```
$ docker run -ti --name c1 --blkio-weight 300 ubuntu:14.04 /bin/bash
$ docker run -ti --name c2 --blkio-weight 600 ubuntu:14.04 /bin/bash
```

例えば、次のようにして２つのコンテナで同時にブロック IO を確認できます。

```
$ time dd if=/mnt/zerofile of=test.out bs=1M count=1024 oflag=direct
```

２つのコンテナ間の blkio ウエイトの割合により、処理にかかる時間の割合が変わるのが分かるでしょう。

blkio ウエイトの設定は直接 IO（direct IO)のみです。現時点ではバッファ IO(buffered IO)をサポートしていません。

グループの追加

--group-add: 実行時のグループを追加

Docker コンテナのプロセスを実行できるのは、デフォルトでは、補助的なグループに所属しているユーザのみです（訳者注：docker グループに所属するユーザ)。グループを更に追加したい場合は、このフラグを使います。

```
$ docker run --rm --group-add audio --group-add nogroup --group-add 777 busybox id
uid=0(root) gid=0(root) groups=10(wheel),29(audio),99(nogroup),777
```

実行時の権限、Linux 機能、LXC 設定

--cap-add: Linux ケーパビリティの追加
--cap-drop: Linux ケーパビリティの削除（ドロップ）

--privileged=false: コンテナに拡張権限を与える
--device=[]: --privileged（特権）フラグが無いコンテナ内でもデバイスの実行を許可
--lxc-conf=[]: カスタム lxc オプションの追加

Docker 1.10 以降では、デフォルトの seccomp プロフィールでは、コンテナに対して --cap-add を指定しても、システムコールをブロックします。このような場合に私たちが推奨するのは、私たちの デフォルト プロフィールを元に書き換える方法です。あるいはデフォルトの seccomp プロファイルを使いたくないのであれば、実行時に --security-opt=seccomp=unconfined を指定できます。

デフォルトでは、Docker コンテナは「unprivileged」（権限が無い）ため、Docker コンテナの中で Docker デーモンを動かす等ができません。これは、デフォルトのコンテナはあらゆるデバイスに対して接続できないためであり、「privileged」（特権）コンテナのみが全てのコンテナに接続できます（lxc-template.go[*1] と cgroups devices[*2] のドキュメントをご覧ください ）

作業者が docker run --privileged を実行したら、Docker はホスト上の全てのデバイスに対して接続可能になります。この時、AppArmor や SELinux の設定があれば、ホスト上のコンテナ外のプロセスと同じように、ホスト上の同じアクセス権限が与えられた状態で利用可能になります。 --privileged の実行に関する追加情報については、Docker ブログの投稿[*3] をご覧ください。

特定のデバイスに対する許可だけ加えたい時は、 --device フラグが使えます。これを指定したら、 1 つまたは複数のデバイスがコンテナ内から接続できるようになります。

```
$ docker run --device=/dev/snd:/dev/snd ...
```

デフォルトでは、コンテナはデバイスに対して read、write、mknod が可能です。それぞれの --device フラグは、:rwm という 3 つのオプション・セットで上書きできます。

```
$ docker run --device=/dev/sda:/dev/xvdc --rm -it ubuntu fdisk  /dev/xvdc

Command (m for help): q
$ docker run --device=/dev/sda:/dev/xvdc:r --rm -it ubuntu fdisk  /dev/xvdc
You will not be able to write the partition table.

Command (m for help): q

$ docker run --device=/dev/sda:/dev/xvdc:w --rm -it ubuntu fdisk  /dev/xvdc
    crash....

$ docker run --device=/dev/sda:/dev/xvdc:m --rm -it ubuntu fdisk  /dev/xvdc
fdisk: unable to open /dev/xvdc: Operation not permitted
```

--privileged に加え、作業者は --cap-add と --cap-drop を使い、ケーパビリティに対する詳細な制御が可能になります。デフォルトでは、Docker はデフォルトのケーパビリティ一覧を保持しています。次の表は、追加・削除可能な Linux ケーパビリティのオプション一覧です。

*1 https://github.com/docker/docker/blob/master/daemon/execdriver/lxc/lxc_template.go

*2 https://www.kernel.org/doc/Documentation/cgroups/devices.txt

*3 http://blog.docker.com/2013/09/docker-can-now-run-within-docker/

ケーパビリティのキー	説明
SETPCAP	プロセスの機能を変更
SYS_MODULE	カーネル・モジュールのロード(load)・アンロード(unload)
SYSRAWIO	ランダム I/O ポート操作(iopl(2)と ioperm(2))
SYS_PACCT	acct(2)を用いたプロセスのスイッチ回数のカウント有無
SYS_ADMIN	システム管理オペレーションの処理範囲
SYS_NICE	プロセスの nice 値(nice(2), setpriority(2)) を上げるのと、任意プロセスに対する nice 値を設定
SYS_RESOURCE	リソース上限の上書き
SYS_TIME	システム・クロック(settimeofday(2), stime(2), adjtimex(2))の設定
SYS_TTY_CONFIG	vhangup(2)を使用。仮想ターミナル上で ioctl(2)オペレーションの関連権限
MKNOD	mknod(2)で特別ファイルを作成
AUDIT_WRITE	カーネル監査（auditing）ログに記録
AUDIT_CONTROL	カーネルの監査（auditing）を有効化。監査フィルタルールの変更や、監査状態やフィルタリング・ルールの読み出し
MAC_OVERRIDE	MAC 設定や状態の変更。Smack LSM 用の実装
MAC_ADMIN	Mandatory アクセス・コントロール（MAC）の上書き。Smack Linux Security Module (LSM) 用の実装
NET_ADMIN	様々なネットワーク関連処理の実施
SYSLOG	特権 syslog(2)処理の実施
CHOWN	ファイルの UID と GID 属性を変更（chown(2)を参照)
NET_RAW	RAW と PACKET ソケットを使用
DAC_OVERRIDE	ファイル音読み書き実行時に迂回し、権限を確認
FOWNER	操作権限の確認時に迂回し、ファイルの UID がシステム上で必要とする UID と一致するか確認
DAC_READ_SEARCH	ファイル読み込み権限の確認を迂回し、ディレクトリの読み込み・実行権限を確認
FSETID	ファイル変更時にユーザ ID とグループ ID を変更しない
KILL	シグナル送信時の権限確認をバイパス
SETGID	プロセス GID を GID 一覧にある任意のものに変更
SETUID	プロセス UID を任意のものに変更
LINUX_IMMUTABLE	FS_APPEND_FL と FS_IMMUTABLE_FL i-node フラグを設定
NET_BIND_SERVICE	ソケットをインターネット・ドメイン権限用のポート（ポート番号は 1024 以下）に割り当て
NET_BROADCAST	ソケットをブロードキャストし、マルチキャストをリッスンする
IPC_LOCK	メモリのロック（mlock(2), mlockall(2), mmap(2), shmctl(2))
IPC_OWNER	System V IPC オブジェクト操作用の権限確認
SYS_CHROOT	chroot(2)を使い、ルート・ディレクトリを変更
SYS_PTRACE	ptrace(2)を使い、任意のプロセスをトレース
SYS_BOOT	reboot(2)と kexec_load(2)を使い、後の処理用にリブートと新しいカーネルを読み込み
LEASE	任意のファイルのリースを確立（詳細は fcntl(2) ）
SETFCAP	ファイルの機能を設定
WAKE_ALARM	システムを起動する何らかのトリガ
BLOCK_SUSPEND	ブロック・システムをサスペンドする機能

より詳細なリファレンス情報は Linux man ページの capabilities(7)[*1] をご覧ください。

作業者は全ての機能を有効化するために ALL 値を使えますが 、MKNOD だけ除外したい時は次のようにします。

```
$ docker run --cap-add=ALL --cap-drop=MKNOD ...
```

*1 http://linux.die.net/man/7/capabilities

ネットワーク・スタックとやりとりするには、 --privileged を使う替わりに、ネットワーク・インターフェースの変更には --cap-add=NET_ADMIN を使うべきでしょう。

```
$ docker run -t -i --rm  ubuntu:14.04 ip link add dummy0 type dummy
RTNETLINK answers: Operation not permitted
$ docker run -t -i --rm --cap-add=NET_ADMIN ubuntu:14.04 ip link add dummy0 type dummy
```

FUSE を基盤とするファイルシステムをマウントするには、 --cap-add と --device の両方を使う必要があります。

```
$ docker run --rm -it --cap-add SYS_ADMIN sshfs sshfs sven@10.10.10.20:/home/sven /mnt
fuse: failed to open /dev/fuse: Operation not permitted
$ docker run --rm -it --device /dev/fuse sshfs sshfs sven@10.10.10.20:/home/sven /mnt
fusermount: mount failed: Operation not permitted
$ docker run --rm -it --cap-add SYS_ADMIN --device /dev/fuse sshfs
# sshfs sven@10.10.10.20:/home/sven /mnt
The authenticity of host '10.10.10.20 (10.10.10.20)' can't be established.
ECDSA key fingerprint is 25:34:85:75:25:b0:17:46:05:19:04:93:b5:dd:5f:c6.
Are you sure you want to continue connecting (yes/no)? yes
sven@10.10.10.20's password:
root@30aa0cfaf1b5:/# ls -la /mnt/src/docker
total 1516
drwxrwxr-x 1 1000 1000   4096 Dec  4 06:08 .
drwxrwxr-x 1 1000 1000   4096 Dec  4 11:46 ..
-rw-rw-r-- 1 1000 1000     16 Oct  8 00:09 .dockerignore
-rwxrwxr-x 1 1000 1000    464 Oct  8 00:09 .drone.yml
drwxrwxr-x 1 1000 1000   4096 Dec  4 06:11 .git
-rw-rw-r-- 1 1000 1000    461 Dec  4 06:08 .gitignore
....
```

Docker デーモンを lxc 実行ドライバを使って起動する時（docker daemon --exec-driver=lxc)、作業者は 1 つまたは複数の LXC オプションを --lxc-conf パラメータで指定できます。これにより、lxc-template.go[*1] にある新しいパラメータの追加や既存のパラメータ上書きが可能です。将来的には、Docker ホストによっては LXC が使えなくなる可能性があるため、注意が必要です。そのため、特定の実装に関する設定操作のためには、LXC の直接操作に慣れておいた方が良いでしょう。

Docker デーモンに管理されているコンテナに対して、--lxc-conf を使いコンテナの設定を変更可能です。しかし Docker デーモンは変更が施されたことを把握できないため、自分自身で管理上の不一致を解決する必要があります。例えば、 --lxc-conf でコンテナの IP アドレスを設定しても、コンテナ内の /etc/hosts ファイルには反映されません。

ログ記録ドライバ（--log-driver）

Docker デーモンはコンテナごとに異なったログ記録ドライバを指定できます。コンテナのログ記録ドライバを指定するには、docker run コマンドで --log-driver=VALUE を指定します。以下のオプションがサポートされています。

*1 https://github.com/docker/docker/blob/master/daemon/execdriver/lxc/lxc_template.go

none	コンテナのログ記録ドライバを無効化します。このドライバでは docker logs が機能しません。
json-file	Docker に対応するデフォルトのログ記録ドライバです。ファイルに JSON メッセージを書き込みます。このドライバに対するオプションはありません。
syslog	Docker に対応する Syslog ログ記録ドライバです。ログのメッセージを syslog に書き込みます。
journald	Docker に対応する Journald ログ記録ドライバです。ログのメッセージを journald に書き込みます。
fluentd	シ Docker に対応する Fluentd ログ記録ドライバです。ログ・メッセージを fluentd に書き込みます（forward input）。
awslogs	Docker に対応する Amazon CloudWatch Logs ロギング・ドライバです。ログ・メッセージを Amazon CloudWatch Logs に書き込みます。

docker logs コマンドが使えるのは json-file と journald ログ記録ドライバのみです。ログ記録ドライバの詳細な情報については ログ記録ドライバの設定 をご覧ください。

Dockerfile イメージのデフォルトより優先

開発者は Dockerfile を使ってイメージ構築時やコミット時、対象イメージを使ったコンテナを起動時に有効な各種パラメータを、開発者自身が設定できます。

実行時に 4 つのコマンド FROM、MAINTAINER、RUN、ADD は上書きできません。それ以外のコマンド全ては docker run で上書きできます。開発者が Dockerfile で個々の命令を設定していたとしても、作業者はその設定を上書きして操作できます。

- CMD（デフォルトのコマンドかオプション）
- ENTRYPOINT（実行時に処理するデフォルトのコマンド）
- EXPOSE （受信用のポート）
- ENV（環境変数）
- VOLUME（共有ファイルシステム）
- USER
- WORKDIR

CMD（デフォルトのコマンドかオプション）

Docker コマンドラインでのオプションコマンドを取り消します。

```
$ docker run [オプション] イメージ[:タグ|@DIGEST] [コマンド] [引数...]
```

このコマンドは様々なオプションを指定します。イメージの作者が Dockerfile の CMD 命令を使い、デフォルトのコマンドを既に設定している場合があるためです。作業者（イメージからコンテナを実行する人）は、CMD 命令を上書きして新しいコマンドを実行します。

イメージに ENTRYPOINT も指定されていれば、CMD やコマンドは ENTRYPOINT に対する引数となります。

ENTRYPOINT（実行時に処理するデフォルトのコマンド）

```
--entrypoint="": Overwrite the default entrypoint set by the image
```

イメージの ENTRYPOINT はコマンドと似ています。これはコンテナを開始する時に実行するコマンドを指定しているためです。しかし、こちらは（意図的に）上書きを難しくしています。ENTRYPOINT が提供するのは、コンテナ

自身が持つデフォルトの特性や振る舞いです。そのため ENTRYPOINT を指定しておけば、コンテナ実行時、あたかもコンテナ自身をバイナリのようにして実行可能です。その場合は、デフォルトのオプションを持っているでしょうし、あるいは自分でコマンドを指定してオプションを指定することも可能です。しかし、時々オペレータはコンテナの中で何らかのコマンドを実行したい場合もあるでしょう。例えば、デフォルトの ENTRYPOINT の代わりに、自分で ENTRYPOINT を新たに指定したい場合です。次の例はコンテナ上でシェルを実行するものであり、同様に何らかのもの（/usr/bin/redis-server のように）を自動的に起動できます。

```
$ docker run -i -t --entrypoint /bin/bash example/redis
```

あるいは、次の２つの例は ENTRYPOINT に更にパラメータを渡すものです。

```
$ docker run -i -t --entrypoint /bin/bash example/redis -c ls -l
$ docker run -i -t --entrypoint /usr/bin/redis-cli example/redis --help
```

EXPOSE （露出用のポート）

run コマンドには、コンテナのネットワークに対応する以下のオプションがあります。

```
--expose=[]: コンテナ内のポートまたはポート範囲を露出する
             これらは「EXPOSE」命令の露出ポートに追加する
-P=false   : 全ての露出ポートをホスト側インターフェースに公開する
-p=[]      : コンテナのポートまたはポート範囲をホスト側に公開する
               書式: ip:ホスト側ポート:コンテナ側ポート | ip::コンテナ側ポート | ホスト側ポート:コンテナ側ポート | コンテナ側ポート
               ホスト側ポートとコンテナ側のポートは、どちらもポート範囲を指定可能です。
               両方で範囲を指定した時は、コンテナ側のポート範囲とホスト側のポート範囲が
               一致する必要があります。例：
                   -p 1234-1236:1234-1236/tcp

               ホスト側のポート範囲しか指定しない時は、コンテナ側ポートが範囲になるとは限りません。
               このような場合、コンテナ側で公開されるポートはホスト側のポート範囲のいずれかです。
               （例 ``-p 1234-1236:1234/tcp`` ）

               （実際の割り当てを確認するには ``docker port`` を使う）

--link="" : 他のコンテナに対するリンクを追加（<名前 or id>:エイリアス or <名前 or id>）
```

イメージの開発者は、EXPOSE 命令以外のネットワーク機能に関する管理は行えません。EXPOSE 命令が定義するのは、サービスが初期化時に提供する受信用ポートです。このポートはコンテナの中のプロセスが利用可能にします。作業者は --expose オプションを使い、公開用ポートを追加できます。

コンテナの内部ポートを露出（expose）するために、作業者はコンテナ実行時に -P か -p フラグを使えます。公開用のポートはホスト上でアクセス可能であり、そのポートはホストに到達可能なクライアントであれば誰でも利用できます。

-P オプションはホスト・インターフェース上に全てのポートを公開します。Docker は公開されたポートを、ホスト側のポートに対してランダムに拘束(bind)します。このポートの範囲を エフェメラル・ポート範囲(ephemeral port range）と呼び、/proc/sys/net/ipv4/ip_local_port_range によって定義されています。 -p フラグを使えば、特定のポートやポートの範囲を割り当てます。

コンテナ内のポート番号（サービスがリッスンしているポート番号）は、コンテナの外に露出するポート番号（クライアントが接続する番号）と一致させる必要がありません。例えば、コンテナ内の HTTP サービスがポート 80 をリッスンしているとします（そして、イメージ開発者は Dockerfile で EXPOSE 80 を指定しているでしょう)。実

行する時に、ホスト側のポート 42800 以上が使われます。公開用ポートがホスト側のどのポートに割り当てられたかを確認するには、docker port コマンドを使います。

デフォルトのブリッジ・ネットワークにおいて、新しいクライアント・コンテナの起動時にオペレータが --link を指定したら、クライアント・コンテナはプライベートなネットワーク・インターフェースを経由して公開ポートにアクセスできます。Docker ネットワーク概要 にあるユーザ定義ネットワーク上で --link を指定したら、コンテナをリンクするためのエイリアス名を作成します。

ENV（環境変数）

新しいコンテナの作成時、Docker は以下の環境変数を自動的に設定します。

変数	値
HOME	USER の値を元にして指定
HOSTNAME	コンテナに関連づけるホスト名
PATH	/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin のような一般的なディレクトリを含む
TERM	コンテナが疑似ターミナル（pseudo-TTY）を割り当

更に、オペレータはコンテナに対して**環境変数の組み合わせ**を -e フラグで追加できます。先ほど言及した環境変数や、開発者が Dockerfile の中で ENV で定義済みの環境変数を上書きできます。

```
$ docker run -e "deep=purple" --rm ubuntu /bin/bash -c export
declare -x HOME="/"
declare -x HOSTNAME="85bc26a0e200"
declare -x OLDPWD
declare -x PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
declare -x PWD="/"
declare -x SHLVL="1"
declare -x container="lxc"
declare -x deep="purple"
```

似たようなものとして、オペレータは -h で hostname （ホスト名）も定義できます。

TMPFS（tmfps ファイルシステムのマウント）

--tmpfs=[]: tmpfs マウントを作成: コンテナ側ディレクトリ[:<オプション>],
オプションは Linux コマンドの 'mount -t tmpfs -o' オプションと同一形式

この例では、オプションで rw、noexec、nosuid、size=65536k、コンテナに対して空の tmpfs をマウントしています。

```
$ docker run -d --tmpfs /run:rw,noexec,nosuid,size=65536k my_image
```

VOLUME（共有ファイルシステム）

-v=[]: バインド（拘束）マウントを作成: [ホスト側ディレクトリ:]コンテナ側ディレクトリ[:<オプション>]
オプションはカンマ区切りで [rw|ro] または [z|Z]、[[r]shared|[r]slave|[r]private]、
そして [nocopy] から選択

host-src は絶対パスもしくは名前を値にします。

rw（読み書き）または ro（読み込み専用）の指定が無ければ、読み書き可能なモードでマウントします。

nocopy モードを指定したら、コンテナ内に要求したボリューム・パスに対して、ボリュームを保存している場所からの自動コピーを無効にします。名前付きボリュームの場合は、copy がデフォルトのモードです。コピー・モードではバインド（拘束）マウント（bind-mounted）したボリュームに対するコピーをサポートしていません。

--volumes-from="": 指定したコンテナにある全てのボリュームをマウント

Ubuntu Utopic 14.10 と 15.04 には Docker の apt リポジトリが存在しますが、（Docker が）公式にサポートしていないものです。

注釈

Docker デーモンの開始・停止を systemd で管理する場合は、Docker デーモン自身がマウント・プロパゲーション（mount propagation）を管理できるよう、systemd の unit ファイル上で MountFlags というオプションを設定します。このマウントポイントが変更されても、Docker はマウント・プロパゲーションの変更を把握できません。例えば、値を slave としているのであれば、ボリュームのプロパゲーション値に shared や rshared を指定すべきではないでしょう。

ボリューム関連コマンドは コンテナでデータを管理 セクション自身のドキュメントでも複雑なものです。開発者は 1 つまたは複数の VOLUME を作成し、イメージと関連づけることが可能です。しかし、作業者ができるのは、あるコンテナから別のコンテナに対してのみです（あるいは、コンテナからホスト側のボリュームにマウントする場合）。

コンテナ側ディレクトリは /src/docs のように常に絶対パスの必要があります。ホスト側ディレクトリは絶対パスか名前を値に指定できます。ホスト側ディレクトリに絶対パスを指定する場合は、Docker は指定したパスを拘束マウント（bind-mounts）します。名前を指定する場合は、Docker は名前を持つボリュームを作成します。

名前は英数字で始まる必要があり、以降は a-z0-9 、_（アンダースコア）、 .（ピリオド）、 -（ハイフン）が使えます。絶対パスは / （フォアワード・スラッシュ）で始める必要があります。

例えば、ホスト側ディレクトリの値に /foo か foo を指定したとします。 /foo 値を指定した場合は、Docker はホスト上に拘束マウントを作成します。foo を指定したら、Docker は指定された名前でボリュームを作成します。

USER

開発者は Dockerfile の USER 命令を使い、 1 つめのプロセスを実行する時のユーザを定義できます。コンテナ起動時 -u オプションを使えば USER 命令を上書きできます。

-u="", --user="": ユーザ名または UID を指定する命令。オプションでグループ名や GUID を指定

以下は有効な例です。

--user=[user | user:group | uid | uid:gid | user:gid | uid:group]

数値で UID を指定する場合は、0 〜 2147483647 の範囲内の必要があります。

WORKDIR

コンテナ内でバイナリを実行する時、デフォルトの作業用ディレクトリはルート(/) ディレクトリです。しかし開発者は Dockerfile の WORKDIR コマンドを使い、デフォルトの作業用ディレクトリを変更できます。作業者が更に設定を上書きするには、次のようにします。

-w="": コンテナ内の作業用（ワーキング）ディレクトリ

A2.2 Docker コマンド

このセクションは、Docker のコマンドライン・クライアントを使うにあたってのリファレンス情報です。各コマンドのリファレンス・ページにはサンプルもあります。コマンドラインに慣れていなければ、次のセクション「Docker コマンドラインを使う」から読み始めた方が良いでしょう。

Docker デーモンの起動はコマンドラインで行います。Docker デーモンは Docker コンテナに影響を与えるものです。そのため、daemon のリファレンス・ページも読んだ方が望ましいでしょう。

Docker 管理コマンド

- daemon
- info
- inspect
- version

イメージ用コマンド

- build
- commit
- export
- history
- images
- import
- load
- rmi
- save
- tag

コンテナ用コマンド

- attach
- cp
- create
- diff
- events
- exec
- kill
- logs
- pause
- port
- ps
- rename

- restart
- rm
- run
- start
- stats
- stop
- top
- unpause
- wait

Hub ・レジストリ用コマンド

- login
- logout
- pull
- push
- search

ネットワークと接続用コマンド

- network_connect
- network_create
- network_disconnect
- network_inspect
- network_ls
- network_rm

共有データボリューム用コマンド

- volume_create
- volume_inspect
- volume_ls
- volume_rm

A1.2.1 Docker コマンドラインを使う

利用可能なコマンドを確認するには、docker にパラメータを付けずに実行するか、docker help を実行します。

```
$ docker
  Usage: docker [OPTIONS] COMMAND [arg...]
         docker daemon [ --help | ... ]
         docker [ --help | -v | --version ]

    -H, --host=[]: The socket(s) to talk to the Docker daemon in the format of tcp://host:port/path,
```

```
    unix:///path/to/socket, fd://* or fd://socketfd.

      A self-sufficient runtime for Linux containers.

      ...
```

Docker のシステム設定によっては、各 docker コマンドの前に sudo を付ける必要があるかもしれません。docker コマンドを実行する度に sudo を実行しないようするには、システム管理者に docker という Unix グループの作成を、そこへのユーザの追加を依頼ください。

Docker のインストールや sudo 設定については、各オペレーティング・システム向けのインストール方法のセクションをご覧ください。

環境変数

簡単に利用できるように、 docker コマンドラインは以下の環境変数をサポートしています。

- DOCKER_CONFIG クライアントの設定ファイルの場所。
- DOCKER_CERT_PATH 認証鍵ファイルの場所。
- DOCKER_DRIVER 使用するグラフドライバ。
- DOCKER_HOST デーモンのソケット接続先。
- DOCKER_NOWARN_KERNEL_VERSION Docker に対応していない Linux カーネルで警告を出さない。
- DOCKER_RAMDISK ‘ pivot_root ’ を無効に設定。
- DOCKER_TLS_VERIFY Docker で TLS とリモート認証を使う。
- DOCKER_CONTENT_TRUST Docker でイメージの署名・確認用のために Notary 使用時に設定。これは、build、create、pull、push、run で --disable-content-trust=false を実行するのと同等
- DOCKER_CONTENT_TRUST_SERVER Notary サーバが使う URL を指定。デフォルトはレジストリと同じ URL 。
- DOCKER_TMPDIR 一時 Docker ファイルの場所。

Docker は「Go」言語で開発されているので、「Go」ランタイムが利用する環境変数も使えます。特に次のものは便利です。

- HTTP_PROXY
- HTTPS_PROXY
- NO_PROXY

これら Go 言語の環境変数は大文字と小文字を区別しません。変数の詳細については Go 言語の仕様[*1] をご確認ください。

設定ファイル

Docker コマンドラインは、ホームディレクトリ$HOME にある .docker ディレクトリ内に保管されている設定ファイルを、デフォルトで使います。しかし、DOCKER_CONFIG 環境変数や --config コマンドライン・オプションを使い、異なった場所を指定できます。両方が指定された場合は --config オプションが DOCKER_CONFIG 環境変数を上書きします。例：

*1 http://golang.org/pkg/net/http/

```
docker --config ~/testconfigs/ ps
```

ps コマンドの実行時、 ~/testconfigs/ ディレクトリにある設定ファイルで Docker に命令します。

Docker は設定ディレクトリにある対部分のファイルを管理していますので、これらを自分で変更すべきではありません。しかし、docker コマンドの振る舞いを制御するため、config.json を編集できます 。

現在、docker コマンドの挙動を環境変数かコマンドラインのオプションで変更可能です。あるいは、オプションとして config.json を使い、同じように淀宇を設定できます。これらの仕組みを使う場合は、優先順位に気を付ける必要があります。コマンドラインのオプションは環境変数で上書きされ、環境変数は config.json ファイルで指定した項目に上書きされます。

config.json ファイルは複数の属性 JSON エンコーディングで記述します。

HttpHeader 属性は、Docker クライアントがデーモンに対して送信する時、全てのメッセージに含めるヘッダを指定します。Docker は、これらのヘッダを解釈したり理解しようとしません。つまり、単純にメッセージの中に追加するだけです。Docker は設定したヘッダ自身に対する変更を許可しません。

psFormat 属性は docker ps 出力のデフォルトの出力フォーマットを指定します。docker ps コマンドで --format フラグが指定されなければ、Docker クライアントはこの属性を使います。この属性が設定されなければ、クライアントはデフォルトの表フォーマットに戻ります。サポートされている形式を確認するには、docker ps ドキュメントにあるフォーマットのセクションをご覧ください。

コンテナにアタッチ後は、CTRL-p CTRL-q キー・シーケンスで使ってデタッチできます。このデタッチ用キー・シーケンスは detachKeys 属性を使ってカスタマイズできます。カスタマイズでは <シーケンス> 値の属性を指定します。 <シーケンス> の書式は [a-Z] までの文字列をカンマ区切りにしたリストにするか、 ctrl- に以下のいずれかを組み合わせます。

- a-z （小文字のアルファベット文字列）
- @ （アット記号）
- [（左かっこ）
- \\ （２つのバックスラッシュ）
- _ （アンダースコア）
- ^ （キャレット）

Docker クライアントで起動するコンテナ全てにカスタマイズが適用されます。ユーザはコンテナごとにデフォルトのキー・シーケンスを変更可能です。ユーザが指定するには、 --detach-keys フラグを docker attach、docker exec、docker run、docker start コマンドで使います。

imageFormat 属性は docker ps 出力のデフォルトの出力フォーマットを指定します。docker images コマンドで --format フラグが指定されなければ、Docker クライアントはこの属性を使います。この属性が設定されなければ、クライアントはデフォルトの表フォーマットに戻ります。サポートされている形式を確認するには、docker images ドキュメントにあるフォーマットのセクションをご覧ください。

以下は config.json ファイルの記述例です：

```
{
  "HttpHeaders": {
    "MyHeader": "MyValue"
  },
  "psFormat": "table {{.ID}}\\t{{.Image}}\\t{{.Command}}\\t{{.Labels}}"
}
```

Notary

自身で Notary（ノータリー）サーバを使っている場合で、もしも自己証明の証明書や、内部の証明機関を使っているのであれば、docker 設定ディレクトリにある tls/<レジストリの URL>/ca.crt 証明書を置き換える必要があります。

あるいは、自分の証明書を信頼できるようにするためには、自分のシステム上のルート証明機関一覧に証明書を追加する方法もあります。

ヘルプ

ヘルプの一覧を表示するには、単純にコマンドを実行するか、 --help オプションを付けます。

```
$ docker run --help

Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Run a command in a new container

  -a, --attach=[]            Attach to STDIN, STDOUT or STDERR
  --cpu-shares=0             CPU shares (relative weight)
...
```

オプションの種類

1 文字のコマンドラインのオプションは、連結できます。 docker run -i -t --name test busybox sh は、 docker run -it -name test busybox sh に書き換えられます。

ブール値

ブール値のオプションとは -d=false のような形式です。何らかのフラグを設定**しない**場合のデフォルト値は、ヘルプテキストで確認できます。ブール値にフラグ値を指定しなければ、デフォルト値に関係なくフラグは true になります。

例えば、docker run -d を実行すると、値は true になります。そのため、コンテナは「デタッチド」モードとしてバックグラウンドで動作します。

オプションのデフォルトは true（例：docker build --rm=true）ですが、デフォルトではない値を指定するには false を明示します。

```
$ docker build --rm=false .
```

複数回の指定

-a=[] のようなオプションは、コマンドライン上で複数回使えます。例えば、次のようなコマンドです。

```
$ docker run -a stdin -a stdout -i -t ubuntu /bin/bash
$ docker run -a stdin -a stdout -a stderr ubuntu /bin/ls
```

オプションによっては -v オプションのように複雑になる場合もあります。

```
$ docker run -v /host:/container example/mysql
```

pty 実装に限界があるため、 -t と -a stderr オプションを同時に使わないでください。 pty モードの stderr （標準エラー出力）は、単純に stdout （標準出力）になります。

文字列と整数

`--name=""` のように文字が含まれるオプションは、1つしか指定できません。 `-c=0` のように整数の場合も、1つしか指定できません。

A2.3 管理用コマンド

daemon（デーモン）

使い方: docker daemon [オプション]

Linux コンテナの自給自足ランタイム

オプション:

--api-cors-header=""	リモート API の CORS ヘッダをセットする
--authorization-plugin=[]	読み込む認証プログインを指定
-b, --bridge=""	コンテナをネットワーク・ブリッジにアタッチ
--bip=""	ネットワーク・ブリッジ IP の指定
--cgroup-parent=	全てのコンテナの親 cgroup を指定
--cluster-store=""	分散ストレージバックエンドの URL
--cluster-advertise=""	クラスタ上のデーモン・インスタンスのアドレス
--cluster-store-opt=map[]	クラスタのオプションを指定
--config-file=/etc/docker/daemon.json	デーモン設定ファイル
--containerd	containerd ソケットのパス
-D, --debug	デバッグ・モードの有効化
--default-gateway=""	コンテナのデフォルト・ゲートウェイ IPv4 アドレス
--default-gateway-v6=""	コンテナのデフォルト・ゲートウェイ IPv6 アドレス
--dns=[]	DNS サーバの指定
--dns-opt=[]	DNS オプションの指定
--dns-search=[]	DNS の検索に使うドメイン（search domain）
--default-ulimit=[]	コンテナのデフォルト ulimit 設定を指定
--exec-opt=[]	実行時のオプションを指定
--exec-root="/var/run/docker"	実行ステート・ファイルのルート・ディレクトリ
--fixed-cidr=""	IP アドレスの IPv4 サブネットを固定
--fixed-cidr-v6=""	IP アドレスの IPv6 サブネットを固定
-G, --group="docker"	unix ソケット用のグループ
-g, --graph="/var/lib/docker"	Docker 実行時の Docker ルート
-H, --host=[]	接続するデーモンのソケット
--help	使い方を表示
--icc=true	コンテナ内部通信（inter-container communication）を有効化
--insecure-registry=[]	安全ではないレジストリとの通信を有効化
--ip=0.0.0.0	コンテナのポートがデフォルトでバインドする IP
--ip-forward=true	net.ipv4.ip_forward の有効化
--ip-masq=true	IP マスカレードの有効化
--iptables=true	iptables のルール追加の有効化
--ipv6	IPv6 ネットワークの有効化
-l, --log-level="info"	ログ記録レベルの設定
--label=[]	デーモンにキー=バリューのラベルを指定
--log-driver="json-file"	デフォルトのコンテナのログ記録ドライバ
--log-opt=[]	ログドライバのオプションを指定
--mtu=0	コンテナ・ネットワークの MTU を指定
--disable-legacy-registry	レガシーのレジストリには接続しない
-p, --pidfile="/var/run/docker.pid"	デーモン PID ファイル用のパス
--registry-mirror=[]	Docker レジストリの優先ミラー
-s, --storage-driver=""	使用するストレージ・ドライバ
--selinux-enabled	SELinux サポートの有効化
--storage-opt=[]	ストレージ・ドライバのオプションを指定
--tls	TLS を使用、--tlsverify を含む
--tlscacert="~/.docker/ca.pem"	この CA で署名された証明書のみ信頼
--tlscert="~/.docker/cert.pem"	TLS 証明書ファイルのパス

```
--tlskey="~/.docker/key.pem"        TLS 鍵ファイルのパス
--tlsverify                         TLS でリモート認証
--userns-remap="default"            ユーザ名前空間の再マッピングを有効化
--userland-proxy=true               ループバック通信用に userland プロキシを使う
```

[] が付いているオプションは、複数回指定できます。

Docker デーモン（daemon）はコンテナを管理するために常駐するプロセスです。Docker はデーモンもクライアントも同じバイナリを使います。デーモンを起動するには docker daemon を入力します。

デーモンでデバッグ出力を行うには、docker daemon -D を使います。

デーモン・ソケットのオプション

Docker デーモンは **Docker リモート API** を受信できます。３種類のソケット unix、tcp、fd を通します。

デフォルトでは unix ドメイン・ソケット（あるいは IPC ソケット）を /var/run/docker.sock に作成します。そのため、操作には root 権限または docker グループへの所属が必要です。

Docker デーモンにリモートからの接続を考えているのであれば、tcp ソケットを有効にする必要があります。デフォルトのセットアップでは、Docker デーモンとの暗号化や認証機能が無いのでご注意ください。また、安全に使うには内部の HTTP 暗号化ソケットを使うべきです。あるいは、安全なウェブ・プロキシをフロントに準備してください。ポート 2375 をリッスンしている場合は、全てのネットワーク・インターフェースで -H tcp://0.0.0.0:2375 を指定するか、あるいは IP アドレスを -H tcp://192.168.59.103:2375 のように指定します。慣例として、デーモンとの通信が暗号化されていない場合はポート 2375 を、暗号化されている場合はポート 2376 を使います。

HTTP 暗号化ソケットを使う時は、TLS　1.0 以上でサポートされているプロトコル SSLv3 以上をお使いください。以下のバージョンはセキュリティ上の理由によりサポートされていません。

systemd をベースとするシステムでは、Systemd ソケット・アクティベーション[*1] を通し、docker daemon -H fd:// で通信が可能です。 fd:// は大部分のセットアップで動作するため、個々のソケットを docker daemon -H fd://3 のように指定できます。もし指定したソケットが見つからない時は、Docker が終了します。Docker で Systemd ソケット・アクティベーションを使う例は Docker のソース・ツリー[*2] をご覧ください。

Docker デーモンは複数の -H オプションを使い、複数のソケットをリッスンできます。

```
# デフォルトの unix ソケットと、ホスト上の２つの IP アドレスをリッスンする
docker daemon -H unix:///var/run/docker.sock -H tcp://192.168.59.106 -H tcp://10.10.10.2
```

Docker クライアントは DOCKER_HOST 環境変数か -H フラグで接続できるようになります。

```
$ docker -H tcp://0.0.0.0:2375 ps
# あるいは
$ export DOCKER_HOST="tcp://0.0.0.0:2375"
$ docker ps
# どちらも同じです
```

DOCKER_TLS_VERIFY 環境変数が設定してあれば、コマンド実行時に --tlsverify フラグを都度指定するのと同じです。以下はいずれも同じです。

*1 http://0pointer.de/blog/projects/socket-activation.html

*2 https://github.com/docker/docker/tree/master/contrib/init/systemd/

```
$ docker --tlsverify ps
# または
$ export DOCKER_TLS_VERIFY=1
$ docker ps
```

Docker クライアントは HTTP_PROXY、HTTPS_PROXY、NO_PROXY 環境変数を（あるいは小文字でも）使えます。HTTPS_PROXY は HTTP_PROXY よりも上位です。

デーモンのストレージ・ドライバ用オプション

Docker デーモンはイメージ・レイヤ用途に、様々に異なるストレージ・ドライバの利用をサポートします。ドライバは、aufs、devicemapper、btrfs、zfs、overlay です。

最も古いのは **aufs（エーユーエフエス） ドライバ**であり、Linux カーネルに対するパッチ群が基礎になっています。ドライバにはメイン・カーネルにマージされなかったコードも含まれます。そのため、深刻なカーネルのクラッシュを引き起こすのが分かっています。一方で、aufs はコンテナの共有実行と共有ライブラリ・メモリが使える唯一のストレージ・ドライバです。そのため、同じプログラムやライブラリで数千ものコンテナを実行する時は便利な選択でしょう。

devicemapper（デバイスマッパー） ドライバは**シン・プロビジョニング（thin provisioning）**と**コピー・オン・ライト（Copy on Write）**スナップショットを使います。devicemapper の各グラフ（Graph）がある典型的な場所は、/var/lib/docker/devicemapper です。シン（thin）プールは２つのブロックデバイス上に作ります。１つはデータで、もう１つはメタデータです。デフォルトでは、別々のファイルとして自動作成したループバックのマウントを元に、これらのブロック・デバイスを自動的に作成します。セットアップのカスタマイズ方法は、以下にある ストレージ・ドライバのオプションをご覧ください。オプションを使わない設定方法については、jpetazzo/Resizing Docker containers with the Device Mapper plugin [*1] の記事に説明があります。

b t r f s（ビーツリーエフエス） ドライバは docker build が非常に高速です。しかし devicemapper のような、デバイス間の実行メモリを共有しません。使うには docker daemon -s btrfs -g /mnt/btrfs_partition を実行します。

zfs ドライバは btrfs ほど速くありませんが、安定さのためレコードを長く追跡します。Single Copy ARC のおかげで、クローン間の共有ブロックを１度キャッシュします。使うには docker daemon -s zf を指定します。異なる zfs ファイルシステムセットを選択するには、zfs.fsname オプションを ストレージ・ドライバのオプションで指定します。

overlay は非常に高速なユニオン・ファイル・システムです。ようやく Linux カーネル 3.18.0 でメインにマージされました。使うには docker daemon -s overlay を指定します。

前途有望な overlay ですが、機能がまだ若く、プロダクションで使うべきではありません。特に overlay を使うと過度の inode 消費を引き起こします（特にイメージが大きく成長する場合）。また、RPM との互換性がありません。

現時点でサポートされていない btrfs や他のコピー・オン・ライトのファイルシステムは、ext4 パーティション上のみで使うべきです。

ストレージ・ドライバのオプション

個々のストレージドライバは --storage-opt フラグでオプションを設定できます。devicemapper 用のオプションは dm で始まり、zfs 用のオプションは zfs で始まります。

*1 http://jpetazzo.github.io/2014/01/29/docker-device-mapper-resize/

dm.thinpooldev

シン・プール用に使うカスタム・ブロックストレージ・デバイスを指定します。

ブロック・デバイスをデバイスマッパー・ストレージに指定する場合は、lvm を使うシン・プール・ボリュームの作成・管理がベストです。その後、このボリュームは Docker により、イメージまたはコンテナで、排他的なスナップショット用ボリュームを作成するために使われます。

シン・プールの管理を Docker Engine の外で行うため、最も機能豊富な手法をもたらします。Docker コンテナの背後にあるストレージとして、Docker はデバイスマッパーによる シン・プロビジョニングを活用するからです。lvm をベースにしたシン・プール管理機能に含まれるハイライトは、自動もしくはインタラクティブなシン・プールの容量変更のサポートです。動的にシン・プールを変更する機能とは、lvm が シン・プールをアクティブにする時、自動的にメタデータのチェックを行います。

シン・プールが割り当てられなければフェイルバックします。この時、ループバックのファイルが作成されます。ループバックは非常に遅いものですが、ストレージの再設定を行わなくても利用可能になります。プロダクション環境においては、ループバックを使わないよう強く推奨します。Docker Engine デーモンで --storage-opt dm.thinpooldev の指定があるのを確認してください。

使用例：

```
$ docker daemon \
      --storage-opt dm.thinpooldev=/dev/mapper/thin-pool
```

dm.basesize

ベース・デバイス作成時の容量を指定します。これはイメージとコンテナのサイズの上限にあたります。デフォルトの値は 10GB です。シン・デバイスは本質的に「希薄」(sparse) なのを覚えて置いてください。そのため、10GB のデバイスの大半が空っぽで未使用だったとしても、10GB の領域がプールされます。しかしながら、ファイルシステムがより大きなデバイスであれば、空っぽだとしても多くの容量を使う可能性があるでしょう。

以後のイメージや（イメージを元にする）コンテナが利用可能となる新しいベース・デバイス容量を増やしたい場合は、デーモンの再起動で変更できます。

使用例：

```
$ docker daemon --storage-opt dm.basesize=50G
```

これはベース・デバイス容量を 50GB に増やしています。Docker デーモンはこのベース・イメージの容量が 50GB よりも大きくなるとエラーを投げます。ユーザはこのオプションを使ってベース・デバイス容量を拡張できますが、縮小はできません。

システム全体の「ベース」となる空白のファイルシステムに対して、設定値が影響を与えます。これは、既に初期化されているか、取得しているイメージから継承している場合です。とりわけ、この値の変更時には、反映するために追加手順が必要です。

```
$ sudo service docker stop
$ sudo rm -rf /var/lib/docker
$ sudo service docker start
```

使用例：

```
$ docker daemon --storage-opt dm.basesize=20G
```

dm.loopdatasize

この設定はデバイスマッパーのループバックを変更するものです。プロダクションで使うべきではありません。

「データ」デバイスがシン・プール用に使うためのループバック・ファイルの作成時、この容量の指定に使います。デフォルトの容量は 100GB です。ファイルは希薄なため、初期段階ではさほど容量を使いません。

使用例：

```
$ docker daemon --storage-opt dm.loopdatasize=200G
```

dm.loopmetadatasize

この設定はデバイスマッパーのループバックを変更するものです。プロダクションで使うべきではありません。

「メタデータ」デバイスがシン・プール用に使うためのループバック・ファイルの作成時、この容量の指定に使います。デフォルトの容量は 2GB です。ファイルは希薄なため、初期段階ではさほど容量を使いません。

使用例：

```
$ docker daemon – storage-opt dm.loopmetadatasize=4G
```

dm.fs

ベース・デバイスで使用するファイルシステムの種類を指定します。サポートされているオプションは「ext4」と「xfs」です。デフォルトは「xfs」です。

使用例：

```
$ docker daemon --storage-opt dm.fs=ext4
```

dm.mkfsarg

ベース・デバイスの作成時に mkfs に対する追加の引数を指定します。

使用例：

```
$ docker daemon --storage-opt "dm.mkfsarg=-O ^has_journal"
```

dm.mountopt

シン・デバイスをマウントする時に使う、追加マウントオプションを指定します。

使用例：

```
$ docker daemon – storage-opt dm.mountopt=nodiscard
```

dm.datadev

（廃止されました。dm.thinpooldev をお使いください ）

シン・プール用のブロック・デバイスが使うデータを指定します。

デバイスマッパー用のストレージにブロック・デバイスを使う時、datadev と metadatadev の両方がループバック・デバイスを完全に使わないようにするのが理想です。

使用例：

```
$ docker daemon \
      --storage-opt dm.datadev=/dev/sdb1 \
      --storage-opt dm.metadatadev=/dev/sdc1
```

dm.metadatadev

（廃止されました。dm.thinpooldev をお使いください ）

シン・プール用のブロック・デバイスが使うメタデータを指定します。

最も性能の高いメタデータとは、データとは軸が異なる場所にあるものです。SSD を使うのが望ましいでしょう。

新しいメタデータ・プールのセットアップには有効化が必要です。次のように、ゼロ値を使い、始めから 4096 まで空白のメタデータを作ります。

```
$ dd if=/dev/zero of=$metadata_dev bs=4096 count=1
```

使用例：

```
$ docker daemon \
      --storage-opt dm.datadev=/dev/sdb1 \
      --storage-opt dm.metadatadev=/dev/sdc1
```

dm.blocksize

シン・プールで使うカスタム・ブロックサイズを指定します。デフォルトのブロックサイズは 64K です。

使用例：

```
$ docker daemon --storage-opt dm.blocksize=512K
```

dm.blkdiscard

デバイスマッパー・デバイスの削除時に blkdiscard を使うか使わないかを指定します。デフォルトは有効であり、ループバック・デバイスを使っているのであれば、イメージやコンテナ削除時にループバック・ファイルを再希薄化させるために使います。

このループバックを無効にしたら、コンテナの削除時間がより早くなります。しかし、 /var/lib/docker ディレクトリで使用している領域量は、コンテナが削除された時点で使っていた領域を返してしまいます。

使用例：

```
$ docker daemon --storage-opt dm.blkdiscard=false
```

dm.override_udev_sync_check

devicemapper と udev 間における udev 同期確認の設定を上書きします。udev は Linux カーネル用のデバイスマッパーです。

Docker デーモンが udev 同期をサポートしているかどうかは、devicemapper ドライバを使い確認します。

```
$ docker info
[...]
Udev Sync Supported: true
[...]
```

udev 同期サポートが true であれば、devicemapper と udev を組み合わせ、コンテナ向けのデバイスを有効化（activation）・無効化（deactivation）します。

udev 同期サポートが false であれば、devicemapper と udev 間で作成・クリーンアップ時に競合を引き起こします。競合状態の結果、エラーが発生して失敗します（の失敗に関する詳しい情報は docker#4036[*1] をご覧ください。）

docker デーモンの起動時に有効にするには、udev 同期をサポートしているかどうかに拘わらず、dm.override_udev_sync_check を true にします。

```
$ docker daemon --storage-opt dm.override_udev_sync_check=true
```

この値が true の場合、devicemapper はエラーが発生しても簡単に警告を表示するだけで、処理を継続します。

docker デーモンと環境を追跡するという考えは、udev の同期機能をサポートするためのものでした。このトピックに関しては docker#4036 をご覧ください。一方で、既存の Docker デーモンを、サポートされている別の環境に移行する時のフラグとしても使います。

dm.use_deferred_removal

libdm やカーネル・ドライバがサポートしている仕組みがあれば、デバイス削除の遅延を有効化します。

デバイス削除の遅延が意味するのは、デバイスを無効化・非アクティブ化しようとしてもビジー（使用中）であれば、デバイス上で遅延削除が予定されます。そして、最後にデバイスを使っているユーザが終了したら、自動的に削除します。

例えば、コンテナを終了したら、関連づけられているシン・デバイスも削除されます。デバイスが他のマウント名前空間も利用しているの場合は、削除できません。コンテナの終了が成功したら、このオプションが有効であれば、システムがデバイスの遅延削除をスケジュールします。使用中のデバイスが削除できるまで、ループを繰り返すことはありません。

使用例：

```
$ docker daemon --storage-opt dm.use_deferred_removal=true
```

dm.use_deferred_deletion

シン・プール用デバイスの遅延削除を有効化するのに使います。デフォルトでは、シン・プールの削除は同期します。コンテナを削除する前に、Docker デーモンは関連するデバイスを削除します。ストレージ・ドライバがデバイスを削除できなければ、コンテナの削除は失敗し、デーモンはエラーを表示します。

この失敗を避けるには、デバイス遅延削除（deletion）と、デバイス遅延廃止（removal）をデーモンで有効化します。

```
$ docker daemon \
      --storage-opt dm.use_deferred_deletion=true \
      --storage-opt dm.use_deferred_removal=true
```

*1 https://github.com/docker/docker/issues/4036

この２つのオプションが有効であれば、ドライバがコンテナを削除する時にデバイスが使用中でも、ドライバはデバイスを削除対象としてマークします。その後、デバイスが使えなくなったら、ドライバはデバイスを削除します。

通常、安全のためにデフォルトでこのオプションを有効化すべきです。複数のマウント名前空間にまたがり、マウントポイントの意図しないリークが発生した時に役立つでしょう。

dm.min_free_space

シン・プールが新しいデバイスを正常に作成するために必要な最小ディスク空き容量を、パーセントで指定します。チェックはデータ領域とメタデータ領域の両方に適用します。有効な値は 0%~99%です。値を 0%に指定すると空き領域のチェック機構を無効にします。ユーザがオプションの値を指定しなければ、Engin はデフォルト値 10%を用います。

新しいシン・プール用デバイスを作成すると（docker pull 時やコンテナの作成時 ）、すぐに Engine は最小空き容量を確認します。十分な領域が無ければデバイスの作成は失敗し、対象の docker オプションは失敗します。

このエラーから復帰するには、エラーが出なくなるようシン・プール内の空き容量を増やす必要があります。シン・プールかにある同じイメージやコンテナを削除することで、空き容量を増やせます。

LVM（Logical Volume Management；論理ボリューム管理）シン・プールの容量を増やすには、コンテナのシン・プールのボリューム・グループに対する領域を追加します。そうすると、エラーは出なくなります。もしループ・デバイスを使う設定であれば、Engine　デーモンは停止します。この問題を解決するにはデーモンを再起動してループ・ファイルの容量を増やします。

指定例：

```
$ docker daemon --storage-opt dm.min_free_space=10%
```

zfs.fsname

現時点で zfs がサポートしているオプションです。Docker が自身のデータセットとして、どの zfs ファイルシステムを使うか指定します。デフォルトの Docker は、docker グラフ（/var/lib/docker）がある場所を zfs ファイルシステムとして用います。

使用例：

```
$ docker daemon -s zfs --storage-opt zfs.fsname=zroot/docker
```

Docker ランタイム実行オプション

Docker デーモンは OCI[*1] 基準のランタイム（containerd（コンテナディー） デーモンの呼び出し）に基づいています。これに従いながら Linux カーネルの名前空間（namespaces）、コントロール・グループ（cgroups）、SELinux に対するインターフェースとして動作します。

ランタイム用のオプション

ランタイムのオプションは --exec-opt フラグで指定できます。全てのオプションのフラグには、先頭に native が付きます。（現時点では）唯一 native.cgroupdriver オプションを利用可能です。

*1 https://github.com/opencontainers/specs

native.cgroupdriver オプションはコンテナの cgroups 管理を指定します。systemd の cgroupfs で指定可能です。systemd で指定時、対象が利用可能でなければ、システムはエラーを返します。native.cgroupdriver オプションを指定しなければ cgroupfs を使います。

次の例は systemd に cgroupdriver を指定しています。

```
$ sudo docker daemon --exec-opt native.cgroupdriver=systemd
```

このオプション設定は、デーモンが起動した全てのコンテナに対して適用します。

また、Windows コンテナであれば特別な目的のために --exec-opt を使えます。Docker がデフォルトで使うコンテナの分離（isolation）技術を指定します。指定例：

```
$ docker daemon --exec-opt isolation=hyperv
```

この指定は Windows 上のデフォルト分離技術として hyperv を使う指定です。デーモン起動時に分離技術の指定が無ければ、デフォルトでは Windows の分離技術として process を使います。

デーモンの DNS オプション

全ての Docker コンテナ用の DNS サーバを設定するには、docker daemon --dns 8.8.8.8 を使います。

全ての Docker コンテナ用の DNS 検索ドメインを設定するには、docker daemon --dns-search example.com を使います。

安全ではないレジストリ

Docker はプライベート・レジストリが安全か否かを確認します。このセクションでは、レジストリの例としてプライベート・レジストリ（private registry）を使い、プライベート・レジストリが myregistry:5000 で動作しているものとします。

安全なレジストリは、TLS を使い、CA 証明書のコピーが /etc/docker/certs.d/myregistry:5000/ca.crt にあります。安全ではないレジストリとは、TLS を使っていない場合（例：平文の HTTP をリッスン）や、TLS を使っていても Docker デーモンが知らない CA 証明書を使う場合を指します。後者であれば、証明書が /etc/docker/certs.d/myregistry:5000/ 以下に存在しないか、証明書の照合に失敗しています（例：CA が違う）。

デフォルトでは、Docker はローカルにあるレジストリ（以下のローカル・レジストリについてをご覧ください）は安全であるとみなします。Docker はレジストリが安全とみなさない限り、安全ではないレジストリとの通信はできません。安全ではないレジストリと通信できるようにするには、Docker デーモンに --insecure-registry という 2 つの形式のオプションが必要です。

- --insecure-registry myregistry:5000 は、Docker デーモンに対して myregistry:5000 が安全ではないと考えられると伝えます。
- --insecure-registry 10.1.0.0/16 は、Docker デーモンに対して、ドメインの逆引き時、CIDR 構文で記述した対象サブネット上に IP アドレスを持つ全てが安全ではないと伝えます。

このフラグは、複数のレジストリに対して安全ではないと複数回指定できます。

安全ではないレジストリを「安全ではない」と指定しなければ、docker pull、docker push、docker search を実行してもエラーメッセージが帰ってきます。ユーザは安全なレジストリを使うか、あるいは先ほどのように --insecure-registry フラグで Docker デーモンに対して明示する必要があります。

IP アドレスが 127.0.0.0/8 の範囲にあるローカルのレジストリは、Docker 1.3.2 以降、自動的に安全ではないレジストリとしてマークされます。ですが、これを信用するのは推奨しません。将来のバージョンでは変更される可能性があります。

--insecure-registry を有効にするとは、暗号化されていない、あるいは信頼できない通信を可能にします。そのため、ローカル環境でのレジストリ実行には便利でしょう。しかし、セキュリティ上の脆弱性を生み出してしまうため、テスト目的のみで使うべきです。セキュリティを高めるには、 --insecure-registry を有効にするのではなく、信頼できる CA 機関が発行する CA を使うべきです。

過去のレジストリ

--disable-legacy-registry を有効にしたら、Docker は v2 プロトコルをサポートしているデーモンとしか通信しないように強制します。この指定によって、デーモンは v1 レジストリへの push、pull、login を阻止します。例外として、v1 レジストリでも search のみ実行できます。

Docker デーモンを HTTPS_PROXY の背後で実行

LAN の内部で HTTPS プロキシを使う場合、Docker Hub の証明書がプロキシの証明書に置き換えられます。これら証明書を、Docker ホストの設定に追加する必要があります。

1. 各ディストリビューションに対応する ca-certificates パッケージをインストールします。

2. ネットワーク管理者にプロキシの CA 証明書を訊ね、 /etc/pki/tls/certs/ca-bundle.crt に追加します。

3. Docker デーモンに HTTPS_PROXY=http://ユーザ名:パスワード@proxy:port/ docker daemon を付けて起動します。ユーザ名:とパスワード@はオプションです。そして、プロ指揮の認証設定も必要であれば追加します。

これは Docker デーモンのリクエストに対してプロキシと認証の設定を追加しただけです。docker build でコンテナを実行する時は、プロキシを使うために更なる追加設定が必要です。

Ulimits のデフォルト

--default-ulimit を使い、全てのコンテナに対するデフォルトの ulimit オプションを指定できます。これは docker run 時に --ulimit オプションを指定するのと同じです。デフォルトを設定しなければ、ulimit 設定を継承します。 docker run 時に設定しなければ、Docker デーモンから継承します。docker run 時のあらゆる --ulimit オプションは、デフォルトを上書きします。

noproc と ulimit フラグを使う時は注意してください。noproc は Linux がユーザに対して利用可能な最大プロセス数を設定するものであり、コンテナ向けではありません。詳細については、 run リファレンスをご確認ください。

ノードのディスカバリ（検出）

--cluster-advertise オプションは、 ホスト:ポート あるいは インターフェース:ポート の組み合わせを指定します。これは、この特定のデーモン・インスタンスがクラスタに自分自身の存在を伝える（advertising）ために使います。リモートホストに到達するデーモンの情報を、ここに指定します。インターフェースを指定する場合は、実際の Docker ホスト上の IP アドレスも含められます。例えば、docker-machine を使ってインストールする時、典型的なインターフェースは eth1 です。

デーモンはクラスタ内のノードに存在を伝えるため、 libkv[*1] を使います。キーバリュー・バックエンドは相互に TLS をサポートします。デーモンが使用するクライアント TLS の設定は --cluster-store-opt フラグを使い、PEM エンコード・ファイルのパスを指定します。実行例：

*1 https://github.com/docker/libkv/

```
docker daemon \
    --cluster-advertise 192.168.1.2:2376 \
    --cluster-store etcd://192.168.1.2:2379 \
    --cluster-store-opt kv.cacertfile=/path/to/ca.pem \
    --cluster-store-opt kv.certfile=/path/to/cert.pem \
    --cluster-store-opt kv.keyfile=/path/to/key.pem
```

現在サポートされているクラスタ・ストアのオプションは以下の通りです。

kv.cacertfile

頼すべき CA 証明書がエンコードされた PEM のローカル・パスを指定します。

kv.certfile

証明書でエンコードされた PEM のローカル・パスを指定。この証明書はクライアントがキーバリュー・ストアとの通信の証明に使います。

kv.keyfile

秘密鍵がエンコードされた PEM のローカル・パスを指定します。この秘密鍵はクライアントがキーバリュー・ストアと通信時に鍵として使います。

kv.path

キーバリュー・ストアのパスを指定します。指定しなければ、デフォルトの `docker/nodes` を使います。

アクセス認証

Docker のアクセス認証は認証プラグインの拡張であり、組織が組織自身で購入・構築できます。認証プラグイン（authorization plugin）を使うには、Docker `daemon` で `--authorization-plugin=PLUGIN_ID` オプションを使って起動します。

```
docker daemon --authorization-plugin=plugin1 --authorization-plugin=plugin2,...
```

PLUGIN_ID の値とは、プラグイン名かファイルのパスを指定します。どのプラグインを実装するかを決めるのは、名前またはパスです。あなたの Docker 管理者に対して、利用可能なプラグインの情報をお訊ねください。

プラグインをインストール後は、コマンドラインや Docker のリモート API を実行する時、プラグインを許可するか許可しないかを選べます。複数のプラグインをインストールした場合は、最後の 1 つだけが処理されます。

認証プラグインの作成方法については、この Docker ドキュメントの拡張に関するセクションにある 認証プラグイン をご覧ください。

デーモンのユーザ名前空間オプション

Linux カーネルの ユーザ名前空間(user namespace)サポート[*1] はプロセスに対する追加のセキュリティを提供します。これを使えば、コンテナでユーザ ID とグループ ID を使う場合、それをコンテナの外、つまり Docker ホスト上で使うユーザ ID とグループ ID のユニークな範囲を指定できます。これは重要なセキュリティ改善になる可能性があります。デフォルトでは、コンテナのプロセスは root ユーザとして動作しますので、コンテナ内で管理特権（と制限）を持っていることが予想されます。しかし、その影響はホスト上の権限の無い uid に対して割り当てられます。

ユーザ名前空間のサポートを有効化したら、Docker はデーモンが扱うマッピングを作成します。これは、同じ Engine のインスタンス上で実行する全コンテナと対応するものです。マッピングを使い、従属ユーザ（subordinate

*1 http://man7.org/linux/man-pages/man7/user_namespaces.7.html

user）ID と従属グループ ID を活用します。この機能は最近の全ての Linux ディストリビューション上において利用可能です。 --userns-remap パラメータを指定することで、 /etc/subuid と /etc/subguid ファイルがユーザとオプションのグループ用に使われます。このフラグに自分でユーザとグループを指定しなければ、ここでは default が指定されます。default のユーザとは dockremap と言う名前であり、各ディストリビューションの一般的なユーザとグループ作成ツールを使い、 /etc/passwd と /etc/group にエントリが追加されます。

現時点ではデーモンごとに１つしかマッピングしない制約があります。これは Engine インスタンス上で実行している全てのコンテナにまたがる共有イメージ・レイヤを Docker が共有しているためです。ファイルの所有者は、レイヤ内容を共有している全てのコンテナで共通の必要があるため、解決策としては docker pull の処理時、ファイル所有者をデーモンのユーザとグループに割り当てる（マッピングする）ことでした。そのため、イメージ内容をダウンロード後は遅延無くコンテナを起動できました。この設計は同じパフォーマンスを維持するため、docker pull と docker push の実行時には維持されています。

ユーザ名前空間を有効にしてデーモンを起動

ユーザ名前空間のサポートを有効化するには、デーモン起動時に --userns-remap フラグを使います。以下のフォーマット形式が指定できます。

- uid
- uid:gid
- ユーザ名
- ユーザ名:グループ名

整数値の ID を指定したら、有効なユーザ名かグループ名に交換されます。これにより、従属 uid と gid の情報が読み込まれ、指定されたこれらのリソースは ID ベースではなく名前ベースでとなります。 /etc/passwd や /etc/group にエントリが無い数値 ID 情報が指定された場合は、docker は起動せずにエラーを表示します。

Fedora 22 では、ユーザ作成時に範囲を割り当てるために必要な /etc/subuid と /etc/subgid ファイルを、touch コマンドで作成する必要があります。この作業は --usernsremap オプションを有効にする前に行わなくてはいけません。ファイルが存在していれば、ユーザが作成した処理が範囲で処理が適切に行われるよう、デーモンを（再）起動できます。

例：default の Docker ユーザ管理

```
$ docker daemon --userns-remap=default
```

default を指定したら、 Docker は dockermap というユーザ名とグループ名が存在しているかどうか確認し、無ければ作成します。ユーザを作成したら、 Linux ディストリビューションは /etc/subuid と /etc/subgid ファイルの使用をサポートします。これは従属ユーザ ID と従属グループ ID を 65536 まで数える（カウントする）もので、これらは既存のファイルへのエントリをオフセットに使います。例えば、Ubuntu は次のような範囲を作成します。既存の user1 という名前のユーザは、既に 65536 までの範囲を持っています。

```
$ cat /etc/subuid
user1:100000:65536
dockremap:165536:65536
```

もしも、既に自分で行った従属ユーザの設定を使いたい場合は、 `--userns-remap` フラグにユーザ名または UID を指定します。グループがユーザ名と一致しない場合は、同様に `gid` やグループ名も指定します。そうしなければ、従属グループ ID の範囲をシステムが応答する時に、ユーザ名がグループ名として使われます。

subuid/subgid 範囲についての詳細情報

パワーユーザであれば、従属 ID の範囲変更という高度な使い方があります。以下で扱うのは、現在どのようにして Docker デーモンが従属範囲のファイルから範囲を決めているかの定義です。

最も簡単なケースは、ユーザとグループに対する**近接範囲**（**contiguous range**）を 1 つだけ指定する場合です。この例では、Docker はコンテナのプロセスに対し、ホスト側の uid と gid の全てを近接範囲として割り当て（マッピングし）ます。つまり、範囲において一番始めに割り当てるのが root ユーザです。この ID が初期 ID として、（ホスト側）範囲における最後をホスト ID 1 として（コンテナ側に）割り当てます。

先ほど取り上げた `/etc/subuid` の例では、root ユーザに再割り当てする uid は 165536 になります。

システム管理者は単一のユーザまたはグループに対して複数の範囲を設定できます。Docker デーモンは利用可能な範囲から、以下のアルゴリズムに基づき範囲を割り当てます。

1. 特定ユーザに対する範囲のセグメント（区分）が見つかれば、開始 ID を昇順でソートします。

2. セグメントの割り当てには、各セグメントの長さに一致するよう、範囲の値を増やします。そうすると、セグメントの範囲は最も低い数値から始まり、これを root として再割り当てし、あとはホスト側の uid/gid と一致する範囲まで繰り返します。例えば、最小セグメントの ID が 1000 から始まり、長さが 100 としたら、1000 を 0 にマップし（root として再マップ）ます。これを対象セグメントでは 1100 が 100 にマップするまで続けます。次のセグメントは ID 10000 から始まる場合、次は 10000 が 101 にマップし、その長さの分だけ処理します。この処理を対象ユーザのサボーディネート（従属）ファイルに空きセグメントが無くなるまで繰り返します。

3. ユーザ向けのセグメント範囲が無くなった場合は、カーネルで 5 つまで使えるよう制限されているエントリ `/proc/self/uid_map` と `/proc/self/gid_map` が使えます。

コンテナ用のユーザ名前空間を無効化

デーモンでユーザ名前空間を有効にしたら、全てのコンテナはユーザ名前空間が有効な状態で起動します。状況によってはコンテナに対するユーザ名前空間を無効化したい時があるでしょう。例えば、特権コンテナ（privileged container）の起動時です（詳細は「ユーザ名前空間と既知の制限」のセクションをご覧ください ）。これらの高度な機能を使うには、コンテナの `run` `exec` `create` コマンド実行時に `--userns=host` を指定します。このオプションを使えばコンテナの利用者に対するユーザ名前空間の割り当てを完全に無効化します。

ユーザ名前空間と既知の制限

Docker デーモンのユーザ名前空間を有効にした状態では、以下の Docker 標準機能は互換性がありません。

- ホスト・モードにおける PID 名前空間または NET 名前空間（`--pid=host` あるいは`--net=host`）
- `--readonly` コンテナ・ファイルシステム（ユーザ名前空間内において、現在のマウント・ファイルシステムのフラグを変更してリマウントすることは、Linux カーネルの制約によりできません）
- デーモンが知らない／機能を持たない外部ドライバ（ボリュームやグラフ）をユーザ名前空間内で実行
- `docker run` で `--privileged` モードのフラグを指定（また `--userns=host` も指定できません ）

一般的に、ユーザ名前空間は高度な機能であり、他の機能との調整を必要とします。例えば、ホストにボリュームをマウントする時は、ファイルの所有者はユーザもしくは管理者がボリューム・コンテナにアクセスできるよう、あらかじめ調整しておきます。

最後に、ユーザ名前空間に対応したコンテナ・プロセス内の root ユーザとは、多くの管理特権を持っていると考えるかもしれません。ですが、Linux カーネルは内部情報に基づきユーザ名前空間内のプロセスとして制限を施します。現時点で最も注意が必要な制約は mknod の使用です。コンテナ内のユーザ名前空間では root であったとしてもデバイス作成の権限がありません。

その他のオプション

IP マスカレードはコンテナがパブリック IP を持っていなくても、インターネット上の他のマシンと通信するための仕組みです。これにより、インターフェースは複数のネットワーク・トポロジを持ちますが、 --ip-masq=false を使って無効化できます。

Docker は Docker データ・ディレクトリ（/var/lib/docker）と /var/lib/docker/tmp に対するソフトリンクをサポートしています。DOCKER_TMPDIR を使っても、データディレクトリを次のように指定可能です。

```
DOCKER_TMPDIR=/mnt/disk2/tmp /usr/local/bin/docker daemon -D -g /var/lib/docker \
    -H unix:// > /var/lib/docker-machine/docker.log 2>&1
# あるいは
export DOCKER_TMPDIR=/mnt/disk2/tmp
/usr/local/bin/docker daemon -D -g /var/lib/docker -H unix:// > /var/lib/docker-machine/docker.log 2>&1
```

デフォルトの親 cgroup

--cgroup-parent オプションは、コンテナがデフォルトで使う親 cgroup（cgroup parent）を指定できます。オプションを設定しなければ /docker を fs cgroup ドライバとして使います。また system.slice を systemd cgroup ドライバとして使います。

cgroup はスラッシュ記号（ / ）で始まるルート cgroup の下に作成されますが、他の cgroup は daemon cgroup の下に作成されます。

デーモンが cgroup daemoncgroup で実行されており、--cgroup-parent=/foobar で/sys/fs/cgroup/memory/foobar の中に cgroup を作成したと仮定時、--cgroup-parent=foobar は/sys/fs/cgroup/memory/daemoncgroup/foobar に cgroup を作成します。

systemd cgroup ドライバは --cgroup-parent と異なるルールです。Systemd のリソース階層は、スライス（訳者注：systemd における CPU やメモリなどのリソースを分割する単位のこと）とツリー上でスライスをエンコードする場所の名前で表します。そのため systemd cgroups 用の --cgroup-parent はスライス名と同じにすべきです。名前はダッシュ区切りの名前で構成します。つまりルート・スライスからのスライスに対するパスです。例えば --cgroup-parent=user-a-b.slice と指定する意味は、コンテナ用の cgroup を /sys/fs/cgroup/memory/user.slice/user-a.slice/user-a-b.slice/docker-<id>.scope に割り当てます。

これらの指定はコンテナに対しても可能です。docker create と docker run の実行時に --cgroup-parent を使えば、デーモンのオプションで指定した --cgroup-parent よりも優先されます。

デーモン設定ファイル

--config-file オプションを使えば、デーモンに対する設定オプションを JSON 形式で指定できます。このファイルでは、フラグと同じ名前をキーとします。ただし、複数の項目を指定可能なフラグの場合は、キーを複数形で指定します（例：label フラグの指定は labels になります ）。デフォルトは、Linux の場合は /etc/docker/daemon.json にある設定ファイルを Docker が読み込もうとします。Windows の場合は %programdata%\docker\config\daemon.json です。

設定ファイル上のオプションは、フラグで指定するオプションと競合してはいけません。ファイルとフラグが重複したまま docker デーモンを起動しようとしても、どのような値を指定しても、起動に失敗します。例えば、デーモンの起動時にラベルを設定ファイルで定義し、かつ、 --label フラグを指定したら、デーモンは起動に失敗し

ます。

デーモン起動時、ファイルに記述しないオプション項目は無視します。次の例は、利用可能な全てのオプションをファイルに記述したものです。

```
{
     "authorization-plugins": [],
     "dns": [],
     "dns-opts": [],
     "dns-search": [],
     "exec-opts": [],
     "exec-root": "",
     "storage-driver": "",
     "storage-opts": [],
     "labels": [],
     "log-driver": "",
     "log-opts": [],
     "mtu": 0,
     "pidfile": "",
     "graph": "",
     "cluster-store": "",
     "cluster-store-opts": [],
     "cluster-advertise": "",
     "debug": true,
     "hosts": [],
     "log-level": "",
     "tls": true,
     "tlsverify": true,
     "tlscacert": "",
     "tlscert": "",
     "tlskey": "",
     "api-cors-headers": "",
     "selinux-enabled": false,
     "userns-remap": "",
     "group": "",
     "cgroup-parent": "",
     "default-ulimits": {},
       "ipv6": false,
       "iptables": false,
       "ip-forward": false,
       "ip-mask": false,
       "userland-proxy": false,
       "ip": "0.0.0.0",
       "bridge": "",
       "bip": "",
       "fixed-cidr": "",
       "fixed-cidr-v6": "",
       "default-gateway": "",
       "default-gateway-v6": "",
       "icc": false,
       "raw-logs": false,
       "registry-mirrors": [],
       "insecure-registries": [],
       "disable-legacy-registry": false
}
```

設定の再読み込み

いくつかのオプションは設定を反映するために、デーモンのプロセスの再起動を必要とせず、実行中のまま行えます。再読み込みするために、Linux では SIGHUP シグナルを使います。Windows では Global\docker-daemon-config-$PID をキーとするグローバル・イベントを使います。設定ファイルでオプションを変更できますが、指定済みのフラグと競合していなか確認されます。もし設定に重複があれば、デーモンは発生を反映できませんが、実行中のデーモンは止まりません。

現時点で変更可能なオプションは以下の通りです。

- debug：true を設定したら、デーモンをデバッグ・モードにします。
- cluster-store：新しいアドレスにディスカバリ・ストアを読み込み直します。
- cluster-store-opts：ディスカバリ・ストアをお見込む時の新しいオプションを指定します。
- cluster-advertise：再起動後のアドバタイズド・アドレスを指定します。
- labels：デーモンのラベルを新しく設定したものに変えます。

--cluster-store、 --cluster-advertise、 --cluster-store-opts のようなクラスタ設定情報の更新や再読み込みが反映できるのは、これまでに指定しない項目に対してのみです。フラグで --cluster-store を指定しても cluster-advertise を指定していなければ、 cluster-advertise は --cluster-store を一緒に指定しなくても反映します。既に設定済みのクラスタ設定に対して変更を試みたら、設定読み込み時に警告メッセージをログに残します。

info（インフォ）

使い方: docker info [オプション]

システムの広範囲な情報を表示します。

--help 使い方を表示。

例：

```
$ docker -D info
Containers: 14
 Running: 3
 Paused: 1
 Stopped: 10
Images: 52
Server Version: 1.9.0
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 545
 Dirperm1 Supported: true
Execution Driver: native-0.2
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge null host
Kernel Version: 3.19.0-22-generic
OSType: linux
Architecture: x86_64
Operating System: Ubuntu 15.04
CPUs: 24
Total Memory: 62.86 GiB
Name: docker
ID: I54V:OLXT:HVMM:TPKO:JPHQ:CQCD:JNLC:O3BZ:4ZVJ:43XJ:PFHZ:6N2S
Docker Root Dir: /var/lib/docker
Debug mode (client): true
Debug mode (server): true
 File Descriptors: 59
 Goroutines: 159
 System Time: 2015-09-23T14:04:20.699842089+08:00
 EventsListeners: 0
 Init SHA1:
 Init Path: /usr/bin/docker
 Docker Root Dir: /var/lib/docker
 Http Proxy: http://test:test@localhost:8080
 Https Proxy: https://test:test@localhost:8080
WARNING: No swap limit support
Username: svendowideit
Registry: [https://index.docker.io/v1/]
Labels:
 storage=ssd
Insecure registries:
 myinsecurehost:5000
 127.0.0.0/8
```

グローバルな `-D` オプションは 、`docker` コマンドのデバッグ情報を表示します。

issue レポートを送信する時は、私たちがどのような設定がされているか知るため、`docker version` と `docker -D info` をお知らせください。

inspect（インスペクト）

使い方: docker inspect [オプション] コンテナ|イメージ [コンテナ|イメージ...]

コンテナあるいはイメージの低レベル情報を表示

```
-f, --format=""           指定する go テンプレートを使い、出力を整形
--help                    使い方を表示
--type=container|image  JSON を返すイメージまたはコンテナの種類を指定
-s, --size=false          種類がコンテナの場合、合計ファイルサイズを表示
```

デフォルトは、全ての結果を JSON 配列で表示します。フォーマットを指定した場合は、それぞれのテンプレートに従って結果を表示します。

デフォルトは、全ての結果を JSON 配列で表示します。コンテナとイメージが同じ名前を持つ場合は、タイプを指定しなければコンテナの JSON を返します。フォーマットを指定した場合は、それぞれのテンプレートに従って結果を表示します。

フォーマットの詳細については、Go 言語の text/template パッケージ[1] の説明をご覧ください。

例

インスタンスの IP アドレスを取得

ほとんどの場合、通常の適切な処理で JSON フィールドから取得できます。

```
$ docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' $INSTANCE_ID
```

インスタンスの MAC アドレスを取得

ほとんどの場合、通常の適切な処理で JSON フィールドから取得できます。

```
$ docker inspect '{{range .NetworkSettings.Networks}}{{.MacAddress}}{{end}}' $INSTANCE_ID
```

インスタンス用ログのパスを取得

```
$ docker inspect --format='{{.LogPath}}' $INSTANCE_ID
```

バインドしているポート一覧を表示

配列の中をループして、割り当てられている結果を簡単な文字で出力します。

```
$ docker inspect \
--format='{{range $p, $conf := .NetworkSettings.Ports}} {{$p}} -> {{(index $conf 0).HostPort}} {{end}}' \
 $INSTANCE_ID
```

特定のポート割り当てを探す

.Field 構文は数字で始まるフィールド名で使えませんが、テンプレート言語の index 機能では利用可能です。.NetworkSettings.Prots セクションには、内部と外部にあるアドレス/ポートのオブジェクトに対する割り当てリス

*1 http://golang.org/pkg/text/template/

トを含みます。index 0 は、そのオブジェクトの 1 番めの項目です。HostPort フィールドからパブリックアドレスを入手するには、次のようにします。

```
$ docker inspect --format='{{(index (index .NetworkSettings.Ports "8787/tcp") 0).HostPort}}' \
    $INSTANCE_ID
```

設定を取得する

.Field 構文は JSON データを含む場合に使えませんが、テンプレート言語のカスタム json であれば利用可能です。 .config セクションは複雑な JSON オブジェクトですが、JSON 全体を取り込んでから、json を使って設定オブジェクトを JSON に変換します。

```
$ docker inspect --format='{{json .config}}' $INSTANCE_ID
```

バージョン
version

使い方: docker version [オプション]

Docker バージョン情報を表示します。

```
-f, --format=""    指定する go テンプレートを使い、出力を整形
--help             使い方を表示
```

デフォルトでは、全てのバージョン情報を読みやすい形式で表示します。フォーマットを指定したら、特定のテンプレートで処理します。

Go 言語の text/template パッケージにフォーマットの全ての詳細が記載されています。

デフォルトの出力

```
$ docker version
Client:
 Version:      1.8.0
 API version:  1.20
 Go version:   go1.4.2
 Git commit:   f5bae0a
 Built:        Tue Jun 23 17:56:00 UTC 2015
 OS/Arch:      linux/amd64

Server:
 Version:      1.8.0
 API version:  1.20
 Go version:   go1.4.2
 Git commit:   f5bae0a
 Built:        Tue Jun 23 17:56:00 UTC 2015
 OS/Arch:      linux/amd64
```

サーバのバージョンを取得

```
$ docker version --format '{{.Server.Version}}'
1.8.0
```

raw データのダンプ

```
$ docker version --format '{{json .}}'
{"Client":{"Version":"1.8.0","ApiVersion":"1.20","GitCommit":"f5bae0a","GoVersion":"go1.4.2","Os":"l
inux","Arch":"amd64","BuildTime":"Tue Jun 23 17:56:00 UTC
2015"},"ServerOK":true,"Server":{"Version":"1.8.0","ApiVersion":"1.20","GitCommit":"f5bae0a","GoVers
ion":"go1.4.2","Os":"linux","Arch":"amd64","KernelVersion":"3.13.2-gentoo","BuildTime":"Tue Jun 23
17:56:00 UTC 2015"}}
```

A2.4 イメージ用コマンド

build（ビルド）

使い方: docker build [オプション] パス | URL | -

パスにあるソースコードから新しいイメージを構築

```
  --build-arg=[]                  構築時の変数を指定
  --cpu-shares                    CPU 共有（相対ウエイト）
  --cgroup-parent=""              コンテナ用のオプション親 cgroup
  --cpu-period=0                  CPU CFS (Completely Fair Scheduler) 間隔の制限
  --cpu-quota=0                   CPU CFS (Completely Fair Scheduler) クォータの制限
  --cpuset-cpus=""                実行時に許可する CPU。例 `0-3`, `0,1`
  --cpuset-mems=""                実行時に許可するメモリ。例 `0-3`, `0,1`
  --disable-content-trust=true    イメージの認証をスキップ
  -f, --file=""                   Dockerfile の名前（デフォルトは 'PATH/Dockerfile'）
  --force-rm                      常に中間コンテナを削除
  --help                          使い方を表示
  --isolation=""                  コンテナの隔離（独立）技術
  --label=[]                      イメージ用のメタデータを指定
  -m, --memory=""                 構築コンテナのメモリ上限を指定
  --memory-swap=""                整数値の指定はメモリにスワップ値を追加。-1 は無制限スワップを有効化
  --no-cache                      イメージ構築時にキャッシュを使わない
  --pull                          常に新しいイメージのダウンロードを試みる
  -q, --quiet                     構築時の表示を抑制し、成功時はイメージ ID を表示
  --rm=true                       構築に成功したら、全ての中間コンテナを削除
  --shm-size=[]                   `/dev/shm` のサイズ。書式は `<数値><単位>`。 `数値` は `0` 以上。単位は `b` (bytes)、`k` (kilobytes)、 `m` (megabytes)、 `g` (gigabytes) のどれか。単位を省略するとバイトになる。サイズを省略すると `64m` になる。
  -t, --tag=[]                    '名前:タグ' 形式で名前とオプションのタグを指定
  --ulimit=[]                     Ulimit オプション
```

Docker イメージは Dockerfile と**コンテクスト（context）**を使って構築します。構築時のコンテクストとは、特定の**パス**や URL の場所にあるファイルのことです。構築中のステップで、対象コンテクスト内のファイルを参照できます。

URL パラメータは Git リポジトリの場所を指定できます。つまり、リポジトリの内容をコンテクストとして構築できます。システムでリポジトリの再帰的なクローンを作成するには git clone --depth 1 --recursive コマンドを使います。このコマンドはローカルホスト上の一時ディレクトリで実行されます。コマンドが成功したら、ディレクトリが Docker デーモンにコンテクストとして送信されます。ローカルのクローンであれば、ローカルなユーザ認証や VPN などを使うプライベート・リポジトリへのアクセスも可能にします。

Git の URL は、コロン : がコンテクストのセクションを分割する設定に使えます。1つめの場所は Git が調査用に参照します。これはブランチ、タグ、コミット SHA が使えます。2つめの場所はリポジトリ内にあるサブディレクトリであり、構築時のコンテクストとして使われます。

例えば、container ブランチを docker という名称のディレクトリでコマンドを実行するには：

```
$ docker build https://github.com/docker/rootfs.git#container:docker
```

次の表は構築コンテクストで有効なサフィックスの一覧です。

構築構文のサフィックス	コミットで利用	構築コンテクストに利用
myrepo.git	refs/heads/master	/
myrepo.git#mytag	refs/heads/mytag	/
myrepo.git#mybranch	refs/heads/mybranch	/
myrepo.git#abcdef	sha1 = abcdef	/
myrepo.git#:myfolder	refs/heads/master	/myfolder
myrepo.git#master:myfolder	refs/heads/master	/myfolder
myrepo.git#mytagr:myfolder	refs/heads/mytag	/myfolder
myrepo.git#mybranch:myfolder	refs/heads/mybranch	/myfolder
myrepo.git#abcdef:myfolder	sha1 = abcdef	/myfolder

コンテクストを指定する代わりに、Dockerfile の URL や STDIN（標準入力）のファイルをパイプできます。STDIN から Dockerfile をパイプするには：

```
docker build - < Dockerfile
```

STDIN や URL を指定したら、システムはコンテクストを Dockerfile という名称のファイルに置き換えるため、-f および --file オプションは無視されます。今回の例では、コンテクストは指定していません。

デフォルトの docker build コマンドは、構築コンテクストのルートにある Dockerfile を探します。-f および --file オプションは、内容が含まれている代替ファイルのパスを指定します。これは複数のファイル群を使って、複数の構築をする場合に便利です。パスには構築コンテクスト用のファイルが必要です。相対パスを指定する時は、現在のディレクトリに対する相対パスを指定する必要があります。

多くの場合、それぞれの Dockerfile を空のディレクトに入れるのがベストな方法です。それから、ディレクトリ内には Dockerfile の構築に必要なものしか置きません。構築のパフォーマンスを向上するには、.dockerignore ファイルを設置し、特定のファイルやディレクトリを除外する設定が使えます。このファイルを作るための詳しい方法は、.dockerignore ファイルのセクションをご覧ください。

Docker クライアントがデーモンと通信できなければ、構築はキャンセルされます。Docker クライアントで ctrl-c を使うか、何らかの理由により Docker クライアントが停止されても、構築は中断されます。

現時点で中断できるのは構築を「実行中」の段階のみです（pull の中断が実装されるまで）。

戻り値（リターンコード）

構築に成功したら、成功の 0 という戻り値を返します。構築に失敗したら、ゼロ以外の戻り値を返します。

失敗理由に関する情報は STDERR に表示されます。

```
$ docker build -t fail .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 1 : FROM busybox
 ---> 4986bf8c1536
Step 2 : RUN exit 13
 ---> Running in e26670ec7a0a
INFO[0000] The command [/bin/sh -c exit 13] returned a non-zero code: 13
$ echo $?
1
```

Dockerfile リファレンスのセクションも別途ご覧ください。

例

PATH で構築

```
$ 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
Successfully built f52f38b7823e
Removing intermediate container 9c9e81692ae9
Removing intermediate container 02071fceb21b
```

この例では PATH に . を指定しています。このローカルディレクトリにある全てのファイルは tar 化され、Docker デーモンに送られます。PATH が示すのは、Docker デーモンが構築時に使う「コンテクスト」（内容物）としてのファイルを見つけるための場所です。デーモンはリモート上のマシンでも操作できるのを思い出してください。これは、クライアント側（docker build コマンドを実行した場所 ）では Dockerfile は何らパース（解析）されません。つまり、PATH に含まれる全てのファイルが送信されるだけでなく、Dockerfile の ADD 命令で追加した場所も含みます。

ローカルのマシンから Docker デーモンにコンテクストを送信時、docker クライアントには「Sending build context」（構築コンテクストの送信中）とメッセージが表示されます。

構築が完了しても中間コンテナをそのまま維持したい場合は、 --rm=false の指定が必要です。こちらを指定すると構築キャッシュに何もしません。

URL で構築

```
$ docker build github.com/creack/docker-firefox
```

これは GitHub リポジトリのクローンを作成し、クローンしたリポジトリをコンテクストとして利用します。リポジトリのルートにある Dockerfile を、構築時の Dockerfile として使います。git://や git@など、その他の Git リポジトリのスキーマを使っても指定可能です。

- で構築

```
$ docker build - < Dockerfile
```

これはコンテクストを使わずに STDIN から Dockerfile を読み込みます。コンテクストが無く、内容物の無いローカルのディレクトリが Docker デーモンに送信されます。コンテクストがありませんので、Dockerfile の ADD はリモートの URL の参照に使えます。

```
$ docker build - < context.tar.gz
```

これは STDIN から圧縮されたコンテクストを読み込み、イメージを構築しています。サポートしているフォーマットは、bzip2、gzip、xz です。

.dockerignore の使い方

```
$ docker build .
Uploading context 18.829 MB
Uploading context
Step 1 : FROM busybox
 ---> 769b9341d937
Step 2 : CMD echo Hello world
 ---> Using cache
 ---> 99cc1ad10469
Successfully built 99cc1ad10469
$ echo ".git" > .dockerignore
$ docker build .
Uploading context  6.76 MB
Uploading context
Step 1 : FROM busybox
 ---> 769b9341d937
Step 2 : CMD echo Hello world
 ---> Using cache
 ---> 99cc1ad10469
Successfully built 99cc1ad10469
```

この例で表示しているのは、.dockerignore ファイルを使い、コンテクストから.git ディレクトリを除外しています。この効果により、アップロードされるコンテクストの容量を小さくしています。構築時のリファレンス .dockerignore ファイルの作成 に、より詳しい情報があります。

イメージのタグ（-t）

```
$ docker build -t vieux/apache:2.0 .
```

これまでの例のように構築していますが、作成されるイメージに対してタグ付けをしています。リポジトリ名は vieux/apache になり、タグは 2.0 になります。

Dockerfile の指定（-f）

```
$ docker build -f Dockerfile.debug .
```

構築時の命令に Dockerfile ではなく、Dockerfile.debug を使うように呼び出しています。

```
$ docker build -f dockerfiles/Dockerfile.debug -t myapp_debug .
$ docker build -f dockerfiles/Dockerfile.prod  -t myapp_prod .
```

上記のコマンドは、どちらも現在のディレクトリにあるコンテント（ . で場所を指定 ）を使い構築するものです。デバッグ用とプロダクション用で別々の Dockerfile を使いますが、コンテクストは同じです。

```
$ cd /home/me/myapp/some/dir/really/deep
$ docker build -f /home/me/myapp/dockerfiles/debug /home/me/myapp
$ docker build -f ../../../../dockerfiles/debug /home/me/myapp
```

２つの docker build コマンドは同じことをしています。いずれの Dockerfile にも debug ファイルが含まれており、構築コンテクストのルートとして /home/me/myapp を使います。なお注意点として、debug は構築コンテクストのサブディレクトリにあるもので、先ほどのコマンドライン上では指定の必要がありませんでした。

docker build が no such file or directory エラーを返すのは、アップロードすべきコンテクストとしてのファイルやディレクトリが存在しない時です。これは、コンテクストが存在しないか、指定したファイルがホストシステム上に存在していない可能性があります。コンテクストはカレント・ディレクトリ（と、その子ディレクトリ）のみに安全上の理由で制限されています。これはリモートの Docker ホスト上でも、繰り返し構築できるようにするためです。これが ADD ../file が動作しない理由でもあります。

親 cgroup のオプション（--cgroup-parent）

docker build に --cgroup-parent オプションを付けて構築したら、構築時の docker run 実行時に適切なフラグを付けて実行します。

コンテナの ulimit をセット（--ulimit）

docker build に --ulimit オプションを付けて実行したら、コンテナの構築ステップを開始する時、都度 --ulimit フラグの値を設定 します。

構築時の変数を指定（--build-arg）

Dockerfile の ENV 命令を使い、変数を定義できます。これらの値は構築時に一定のものです。しかし、一定の値が必要でない場合もあります。ユーザがイメージを構築するホストによっては、依存性に対する変数が必要になるかもしれません。

良い例が http_proxy や中間ファイルの取得に使うソースのバージョン指定です。ARG 命令は Dockerfile の作者が定義する値であり、ユーザが構築時に --build-arg フラグを指定できます。

```
$ docker build --build-arg HTTP_PROXY=http://10.20.30.2:1234 .
```

このフラグを使うことで、構築時の変数が Dockerfile の RUN 命令で通常の環境変数のように扱えます。それだけでなく、これらの値は ENV のように使えますが、中間ファイルや最終的なイメージでは一定ではありません。

ARG と ENV 命令の詳細については、Dockerfile リファレンス をご覧ください。

commit（コミット）

使い方: docker commit [オプション] コンテナ [リポジトリ[:タグ]]

コンテナの変更を元に新しいイメージを作成

```
-a, --author=""     作者（例 "John Hannibal Smith <hannibal@a-team.com>"）
-c, --change=[]     イメージをコミット時の Dockerfile 命令を追加指定
--help              使い方を表示
-m, --message=""    コミット・メッセージ
-p, --pause=true    コンテナをコミット時に一時停止（pause)する
```

コンテナのファイル変更や設定を、新しいイメージに収容（commit；コミット）するために便利です。これにより、インタラクティブなシェル上でコンテナをデバッグ用に動かしたり、作業中のデータセットを他のサーバに持っていくため出力したりできます。通常は、イメージを管理するためには、文書化されメンテナンスのしやすい Dockerfile を使のが望ましい方法です。

コンテナ内でマウントされているボリュームに含まれるデータは、コミット作業に含まれません。

デフォルトでは、コンテナをコミットする時、その過程のいてイメージをコミットする間は一時的に停止します。これはコミットする糧において、データ破損が発生する可能性を減らします。この動作を理解しているのであれば、--pause オプションを使って無効化もできます。

--cache オプションは Dockerfile の命令でイメージが作られる時のみ適用されます。対応している Dockerfile 命令は、CMD、ENTRYPOINT、ENV、EXPOSE、LABEL、ONBUILD、USER、VOLUME、WORKDIR です。

コンテナのコミット

```
$ docker ps
ID                  IMAGE               COMMAND             CREATED             STATUS
PORTS
c3f279d17e0a        ubuntu:12.04        /bin/bash           7 days ago          Up 25 hours
197387f1b436        ubuntu:12.04        /bin/bash           7 days ago          Up 25 hours
$ docker commit c3f279d17e0a  svendowideit/testimage:version3
f5283438590d
$ docker images
REPOSITORY                      TAG                 ID                  CREATED             SIZE
svendowideit/testimage          version3            f5283438590d        16 seconds ago      335.7 MB
```

新しい設定でコンテナをコミット

```
$ docker ps
ID                  IMAGE               COMMAND             CREATED             STATUS
PORTS
c3f279d17e0a        ubuntu:12.04        /bin/bash           7 days ago          Up 25 hours
197387f1b436        ubuntu:12.04        /bin/bash           7 days ago          Up 25 hours
$ docker inspect -f "{{ .Config.Env }}" c3f279d17e0a
[HOME=/ PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin]
$ docker commit --change "ENV DEBUG true" c3f279d17e0a  svendowideit/testimage:version3
f5283438590d
$ docker inspect -f "{{ .Config.Env }}" f5283438590d
[HOME=/ PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin DEBUG=true]
```

新しい CMD と EXPOSE 命令でコンテナをコミット

```
$ docker ps
ID                  IMAGE               COMMAND             CREATED             STATUS
PORTS
c3f279d17e0a        ubuntu:12.04        /bin/bash           7 days ago          Up 25 hours
197387f1b436        ubuntu:12.04        /bin/bash           7 days ago          Up 25 hours

$ docker commit --change='CMD ["apachectl", "-DFOREGROUND"]' -c "EXPOSE 80" c3f279d17e0a
svendowideit/testimage:version4
f5283438590d

$ docker run -d svendowideit/testimage:version4
89373736e2e7f00bc149bd783073ac43d0507da250e999f3f1036e0db60817c0

$ docker ps
ID                  IMAGE               COMMAND                  CREATED             STATUS
  PORTS
89373736e2e7        testimage:version4  "apachectl -DFOREGROU"  3 seconds ago       Up 2 seconds
80/tcp
c3f279d17e0a        ubuntu:12.04        /bin/bash                7 days ago          Up 25 hours
197387f1b436        ubuntu:12.04        /bin/bash                7 days ago          Up 25 hours
```

エクスポート
export

使い方: docker export [オプション] コンテナ

コンテナのファイルシステム内容を tar アーカイブとして出力

```
  --help              使い方の表示
  -o, --output=""     STDOUT（標準出力）ではなくファイルに書き出し
```

docker export コマンドは、コンテナに関連づけられているボリュームに含まれる内容を出力しません。もしボリュームがコンテナ内にある既存ディレクトリの上位にマウントされている場合は、docker export は配下にあるディレクトリを出力しますが、ボリュームの内容は含みません。

詳細はユーザ・ガイドの「データ・ボリュームのバックアップ、移行、レストア にある、ボリューム上のデータの出力について」のセクションをご覧ください。

例

```
$ docker export red_panda > latest.tar
```

または

```
$ docker export --output="latest.tar" red_panda
```

history（ヒストリ）

使い方: docker history [オプション] イメージ

イメージの履歴を表示

```
  -H, --human=true      サイズと日付を人が読みやすい形式で表示
  --help                使い方の表示
  --no-trunc=false      トランケート（truncate）を出力しない
  -q, --quiet=false     整数値の ID のみ表示
```

docker:latest イメージをどのように構築したかを確認します。

```
$ docker history docker
IMAGE               CREATED             CREATED BY                                      SIZE
      COMMENT
3e23a5875458        8 days ago          /bin/sh -c #(nop) ENV LC_ALL=C.UTF-8            0 B
8578938dd170        8 days ago          /bin/sh -c dpkg-reconfigure locales &&    loc   1.245 MB
be51b77efb42        8 days ago          /bin/sh -c apt-get update && apt-get install    338.3 MB
4b137612be55        6 weeks ago         /bin/sh -c #(nop) ADD jessie.tar.xz in /        121 MB
750d58736b4b        6 weeks ago         /bin/sh -c #(nop) MAINTAINER Tianon Gravi <ad   0 B
511136ea3c5a        9 months ago                                                        0 B
     Imported from -
```

docker:scm イメージが何のコンテナ・ベース・イメージから作られたかを確認します。

```
$ docker history docker:scm
IMAGE               CREATED             CREATED BY                                      SIZE
      COMMENT
2ac9d1098bf1        3 months ago        /bin/bash                                       241.4 MB
     Added Apache to Fedora base image
88b42ffd1f7c        5 months ago        /bin/sh -c #(nop) ADD file:1fd8d7f9f6557cafc7   373.7 MB
c69cab00d6ef        5 months ago        /bin/sh -c #(nop) MAINTAINER Lokesh Mandvekar   0 B
511136ea3c5a        19 months ago                                                       0 B
```

イメージーズ
images

使い方: docker images [オプション] [リポジトリ[:タグ]]

イメージを一覧表示します。

```
-a, --all=false      全てのイメージを表示（デフォルトは中間コンテナを非表示）
--digests=false      digest 値を表示
-f, --filter=[]      指定した状況に応じて出力を整形
--help               使い方の表示
--no-trunc=false     トランケート（truncate）を出力しない
-q, --quiet=false    整数値の ID のみ表示
```

標準の docker image は全てのトップ・レベルのイメージと、リポジトリ・タグ・容量を表示します。

Docker イメージは中間レイヤ（intermediate layer）を持っています。これは再利用性を高め、ディスク容量を減らし、docker build は各ステップをキャッシュするので速度を向上します。デフォルトでは、これらの中間レイヤは表示されません。

SIZE（容量）は、イメージと全ての親イメージの累積した領域です。また、docker save でイメージを作成していた場合、Tar ファイルの内容に含まれるディスク容量です。

イメージ一覧では、複数のリポジトリ名やタグが表示されます。イメージ（IMAGE ID が一致するもの ）ごとの SIZE が複数表示されますが、実際に対象としている容量は 1 つだけです。

直近で作成したイメージから一覧表示

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              77af4d6b9913        19 hours ago        1.089 GB
committ             latest              b6fa739cedf5        19 hours ago        1.089 GB
<none>              <none>              78a85c484f71        19 hours ago        1.089 GB
docker              latest              30557a29d5ab        20 hours ago        1.089 GB
<none>              <none>              5ed6274db6ce        24 hours ago        1.089 GB
postgres            9                   746b819f315e        4 days ago          213.4 MB
postgres            9.3                 746b819f315e        4 days ago          213.4 MB
postgres            9.3.5               746b819f315e        4 days ago          213.4 MB
postgres            latest              746b819f315e        4 days ago          213.4 MB
```

イメージ名とタグで一覧表示

docker images コマンドは、オプションで [リポジトリ[:タグ]] を指定できます。これはイメージ一覧から条件が一致するものだけ表示します。リポジトリはタグを指定しなくても使えますので、docker images で対象となるリポジトリの全イメージのみ表示します。

例えば、「java」リポジトリにあるイメージを表示するには、次のコマンドを実行します。

```
$ docker images java
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
java                8                   308e519aac60        6 days ago          824.5 MB
java                7                   493d82594c15        3 months ago        656.3 MB
java                latest              2711b1d6f3aa        5 months ago        603.9 MB
```

[リポジトリ[:タグ]] 値は「完全一致」の必要があります。つまり、docker images jav は java イメージに一致しません。

リポジトリとタグの両方が指定された場合は、リポジトリとタグが一致するイメージのみ表示します。ローカルにある「java」リポジトリで、タグが「8」のイメージを表示するには、次のように実行します。

```
$ docker images java:8
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
java                8                   308e519aac60        6 days ago          824.5 MB
```

もし一致する [リポジトリ[:タグ]] が無ければ、何も表示しません。

```
$ docker images java:0
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
```

長いイメージ ID で全てを表示

```
$ docker images --no-trunc
REPOSITORY                    TAG                 IMAGE ID
                     CREATED             SIZE
<none>                        <none>
sha256:77af4d6b9913e693e8d0b4b294fa62ade6054e6b2f1ffb617ac955dd63fb0182   19 hours ago        1.089 GB
committest                    latest
sha256:b6fa739cedf5ea12a620a439402b6004d057da800f91c7524b5086a5e4749c9f   19 hours ago        1.089 GB
<none>                        <none>
sha256:78a85c484f71509adeaace20e72e941f6bdd2b25b4c75da8693efd9f61a37921   19 hours ago        1.089 GB
docker                        latest
sha256:30557a29d5abc51e5f1d5b472e79b7e296f595abcf19fe6b9199dbbc809c6ff4   20 hours ago        1.089 GB
<none>                        <none>
sha256:0124422dd9f9cf7ef15c0617cda3931ee68346455441d66ab8bdc5b05e9fdce5   20 hours ago        1.089 GB
<none>                        <none>
sha256:18ad6fad340262ac2a636efd98a6d1f0ea775ae3d45240d3418466495a19a81b   22 hours ago        1.082 GB
<none>                        <none>
sha256:f9f1e26352f0a3ba6a0ff68167559f64f3e21ff7ada60366e2d44a04befd1d3a   23 hours ago        1.089 GB
tryout                        latest
sha256:2629d1fa0b81b222fca63371ca16cbf6a0772d07759ff80e8d1369b926940074   23 hours ago        131.5 MB
<none>                        <none>
sha256:5ed6274db6ceb2397844896966ea239290555e74ef307030ebb01ff91b1914df   24 hours ago        1.089 GB
```

イメージの digest 値を表示

v2 以降の形式を使うイメージには、digest と呼ばれる識別子が割り振られます。イメージ生成後に変更が加えられなければ、digest 値は変更されていないと考えられます。全ての digest 値を表示するには、 --digests フラグを使います。

```
$ docker images --digests
REPOSITORY                          TAG                 DIGEST
                             IMAGE ID            CREATED             SIZE
localhost:5000/test/busybox         <none>
sha256:cbbf2f9a99b47fc460d422812b6a5adff7dfee951d8fa2e4a98caa0382cfbdbf   4986bf8c1536        9 weeks
ago         2.43 MB
```

2.0 レジストリに対して送信 (push) や取得 (pull) する場合は、push と pull コマンドの出力にイメージの digest 値も含まれます。digest 値を使っても pull できます。digest 値が使えるのは create、run、rmi の各コマンドと、Dockerfile のイメージを参照する FROM でも同様です。

フィルタリング

フィルタリング・フラグ（ -f と --filter）の形式は「key=value」です。複数のフィルタを使う時は、複数のフラグを使います（例： --filter "foo=bar" --filter "bif=baz" ）。

現在サポートされているフィルタ：

- ダングリング（宙ぶらりんな状態）なイメージ （ブール値： true か false ）
- ラベル（ label=<key> か lavel=<key>=<value> ）

タグ付けされていないイメージ（dangling）

```
$ docker images --filter "dangling=true"

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              8abc22fbb042        4 weeks ago         0 B
<none>              <none>              48e5f45168b9        4 weeks ago         2.489 MB
<none>              <none>              bf747efa0e2f        4 weeks ago         0 B
<none>              <none>              980fe10e5736        12 weeks ago        101.4 MB
<none>              <none>              dea752e4e117        12 weeks ago        101.4 MB
<none>              <none>              511136ea3c5a        8 months ago        0 B
```

これはタグ付けされておらず、イメージ・ツリーから離れた（中間レイヤではない）イメージを表示します。これらのタグが無いイメージは、イメージを使って新しく構築しようとしてもリポジトリ:タグの形式が利用できないため、その場合はイメージ ID を使います。コンテナが利用中であれば、イメージを削除しようとしても警告が表示されます。バッチ処理でクリーンアップする時に、このフラグが使えます。

docker rmi に対応するには：

```
$ docker rmi $(docker images -f "dangling=true" -q)

8abc22fbb042
48e5f45168b9
bf747efa0e2f
980fe10e5736
dea752e4e117
511136ea3c5a
```

タグ付けされていないイメージでも、何らかのコンテナが使用中であれば Docker は警告を表示します。

ラベル付けされたイメージ

label フィルタは、label そのものが一致するイメージか、ラベルの値に一致する場合に表示します。

次のフィルタは com.example.version に一致するラベルだけでなく、その値にも適用されます。

```
$ docker images --filter "label=com.example.version"

REPOSITORY          TAG                 IMAGE ID            CREATED              SIZE
match-me-1          latest              eeae25ada2aa        About a minute ago   188.3 MB
match-me-2          latest              eeae25ada2aa        About a minute ago   188.3 MB
```

次のフィルタは com.example.version ラベルと 1.0 値に一致するイメージを表示します。

```
$ docker images --filter "label=com.example.version=1.0"
REPOSITORY          TAG                 IMAGE ID            CREATED              SIZE
match-me            latest              eeae25ada2aa        About a minute ago   188.3 MB
```

次の例は、0.1 値を持つものをフィルタしますが、一致するものが無かったため、何も表示されません。

```
$ docker images --filter "label=com.example.version=0.1"
REPOSITORY          TAG                 IMAGE ID            CREATED              SIZE
```

インポート
import

使い方: docker import ファイル|URL|- [リポジトリ[:タグ]]

空のファイルシステム・イメージを作成し、tar ボールの内容を読み込む（import）
tar ボールの形式は（.tar, .tar.gz, .tgz, .bzip, .tar.xz, .txz）
オプションで、取り込み後にタグ付け

```
-c, --change=[]     イメージ読み込み時に Dockerfile の命令を追加変更
--help              使い方を表示
-m, --message=      イメージを取り込み時、コミット用のメッセージを設定
```

URL か - （ダッシュ）の指定、あるいは STDIN（標準入力）から直接データを取り込みます。URL はアーカイブ（ .tar、.tar.gz、.tgz、.bzip、.tar.xz、txz）に含まれる圧縮ファイルシステムや、Docker ホスト上の個々のファイルを指定します。アーカイブを指定したら、Docker はコンテナの / （ルート）以下の相対パスとして展開します。個々のファイルを指定する場合、ホスト上のフルパスを指定する必要があります。リモートの場所から import する場合、URI の形式は、http://か https://プロトコルで始まる必要があります。

--change オプションが適用されるのは、Dockerfile 命令でイメージの作成時です。サポートされている Dockerfile の命令は、CMD ENTRYPOINT ENV EXPOSE ONBUID USER VOLUME WORKDIR です。

例

リモートから取り込む

新しいタグ付けされていないイメージを作成します。

```
$ docker import http://example.com/exampleimage.tgz
```

ローカル・ファイルを取り込む

STDIN のパイプ経由で Docker に取り込みます。

```
$ cat exampleimage.tgz | docker import - exampleimagelocal:new
```

コミット・メッセージを付けて取り込みます。

```
$ cat exampleimage.tgz | docker import --message "New image imported from tarball" -
exampleimagelocal:new
```

ローカルのアーカイブから取り込みます。

```
$ docker import /path/to/exampleimage.tgz
```

ローカルのディレクトリを取り込む

```
$ sudo tar -c . | docker import - exampleimagedir
```

新しい設定でローカルのディレクトリを取り込む

```
$ sudo tar -c . | docker import --change "ENV DEBUG true" - exampleimagedir
```

この例では sudo を使っているのに気を付けてください。これは、tar アーカイブを処理する時にファイル（特に

root）の所有権を保持するためです。root でなければ（あるいは sudo を使わなければ)、tar の利用時に権限を得られない可能性があります。

load（ロード）

使い方: docker load [オプション]

tar アーカイブまたは STDIN （標準入力）からイメージを読み込む

```
  --help              使い方の表示
  -i, --input=""      STDIN ではなく、tar アーカイブ・ファイルから読み込む。tar は gzip、bzip、 xz で圧縮されている場合がある
  -q, --quiet         読み込み時に出力を抑制。オプションを指定しなければ、進捗バーを表示
```

ファイルか標準入力のストリームから tar 化されたリポジトリを読み込み（load）ます。イメージとタグを復元します。

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
$ docker load < busybox.tar.gz
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
busybox             latest              769b9341d937        7 weeks ago         2.489 MB
$ docker load --input fedora.tar
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
busybox             latest              769b9341d937        7 weeks ago         2.489 MB
fedora              rawhide             0d20aec6529d        7 weeks ago         387 MB
fedora              20                  58394af37342        7 weeks ago         385.5 MB
fedora              heisenbug           58394af37342        7 weeks ago         385.5 MB
fedora              latest              58394af37342        7 weeks ago         385.5 MB
```

rmi

使い方: docker rmi[*1] [オプション] イメージ [イメージ...]

１つまたは複数のイメージを削除

```
  -f, --force          イメージの共生削除
  --help               使い方の表示
  --no-prune           タグが無い親イメージを削除しない
```

ショート ID かロング ID、タグ、digest を使ってイメージを削除できます。イメージがタグによって参照されている場合、イメージを削除する前にそれらの削除が必要です。Digest の参照はイメージのタグを削除する時、自動的に削除されます。

```
$ docker images
REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
test1                     latest              fd484f19954f        23 seconds ago      7 B (virtual 4.964
MB)
test                      latest              fd484f19954f        23 seconds ago      7 B (virtual 4.964
MB)
test2                     latest              fd484f19954f        23 seconds ago      7 B (virtual 4.964
MB)

$ docker rmi fd484f19954f
Error: Conflict, cannot delete image fd484f19954f because it is tagged in multiple repositories, use -f to
force
2013/12/11 05:47:16 Error: failed to remove one or more images

$ docker rmi test1
Untagged: test1:latest
$ docker rmi test2
Untagged: test2:latest

$ docker images
REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
test                      latest              fd484f19954f        23 seconds ago      7 B (virtual 4.964
MB)
$ docker rmi test
Untagged: test:latest
Deleted: fd484f19954f4920da7ff372b5067f5b7ddb2fd3830cecd17b96ea9e286ba5b8
```

-f フラグでイメージのショート ID かロング ID を指定したら、このコマンド対象の ID に一致するイメージは全てのタグを外し、削除されます。

```
$ docker images
REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
test1                     latest              fd484f19954f        23 seconds ago      7 B (virtual 4.964
MB)
test                      latest              fd484f19954f        23 seconds ago      7 B (virtual 4.964
MB)
test2                     latest              fd484f19954f        23 seconds ago      7 B (virtual 4.964
```

*1 訳者注：rmi とは remove image、つまりイメージ削除コマンドです。

```
MB)

$ docker rmi -f fd484f19954f
Untagged: test1:latest
Untagged: test:latest
Untagged: test2:latest
Deleted: fd484f19954f4920da7ff372b5067f5b7ddb2fd3830cecd17b96ea9e286ba5b8
```

取得したイメージがタグ付けされていなくても、digest を確認できます。

```
$ docker images --digests
REPOSITORY                      TAG       DIGEST
              IMAGE ID        CREATED         SIZE
localhost:5000/test/busybox     <none>
sha256:cbbf2f9a99b47fc460d422812b6a5adff7dfee951d8fa2e4a98caa0382cfbdbf   4986bf8c1536    9 weeks ago
    2.43 MB
```

digest を使ってイメージを削除するには、次のようにします。

```
$ docker rmi
localhost:5000/test/busybox@sha256:cbbf2f9a99b47fc460d422812b6a5adff7dfee951d8fa2e4a98caa0382cfbdbf
Untagged:
localhost:5000/test/busybox@sha256:cbbf2f9a99b47fc460d422812b6a5adff7dfee951d8fa2e4a98caa0382cfbdbf
Deleted: 4986bf8c15363d1c5d15512d5266f8777bfba4974ac56e3270e7760f6f0a8125
Deleted: ea13149945cb6b1e746bf28032f02e9b5a793523481a0a18645fc77ad53c4ea2
Deleted: df7546f9f060a2268024c8a230d8639878585defcc1bc6f79d2728a13957871b
```

セーブ
save

使い方: docker save [オプション] イメージ [イメージ...]

１つまたは複数のイメージを tar アーカイブに保存（デフォルトは STDOUT にストリーム）

```
--help              使い方の表示
-o, --output=""     STDOUT（標準出力）の代わりに、ファイルへ書き込む
```

tar 化されたリポジトリを、標準出力のストリームに出力します。ここには全ての親レイヤが含まれ、全てのタグとバージョンだけでなく、リポジトリ名:タグ が指定されれば、それぞれの引数に応じて出力します。
これはバックアップの作成のために利用でき、再度使うには docker load を指定します。

```
$ docker save busybox > busybox.tar
$ ls -sh busybox.tar
2.7M busybox.tar
$ docker save --output busybox.tar busybox
$ ls -sh busybox.tar
2.7M busybox.tar
$ docker save -o fedora-all.tar fedora
$ docker save -o fedora-latest.tar fedora:latest
```

イメージのリポジトリで、適切なタグを指定する場合にも便利でしょう。

```
$ docker save -o ubuntu.tar ubuntu:lucid ubuntu:saucy
```

タグ
tag

使い方: docker tag [オプション] イメージ[:タグ] [レジストリのホスト/][ユーザ名/]名前[:タグ]

リポジトリ内のイメージにタグ付け

--help 使い方の表示

自分のイメージを名前やタグを使ってグループ化し、 リポジトリを通してイメージを共有するためアップロードできます。

A2.5 コンテナ用コマンド

attach（アタッチ）

使い方: docker attach [オプション] コンテナ

実行中のコンテナにアタッチする

--detach-keys="<sequence>"	エスケープ・キー・シーケンスを設定
--help	使い方の表示
--no-stdin	STDIN（標準入力）にアタッチしない
--sig-proxy=true	受信したシグナルをプロセスに全てプロキシする

docker attach コマンドは、コンテナ ID や名前を使って実行中のコンテナに**アタッチ**（attach；装着/取り付けの意味）します。処理中の出力を表示するだけでなく、インタラクティブ（双方向）の管理もできます。同じコンテナ化されたプロセスに対して、画面を共有する形式として擬似的に複数回のアタッチが可能ですし、デタッチ（detach；分離の意味）したプロセスも迅速に表示できます。

コンテナを停止するには、CTRL-c を使います。このキー・シーケンスはコンテナに対して SIGKILL を送信します。もしも --sig-proxy が true であれば（デフォルト）、CTRL-c はコンテナに対して SIGINT を送信します。CTRL-p CTRL-q キー・シーケンスを使えば、実行中のコンテナからデタッチして離れられます。

コンテナ内で PID 1 として実行しているプロセスは、Linux では特別な扱いがされます。通常の操作では、あらゆるシグナルを無視します。そのため、特別にコード化しない限り、プロセスを SIGINT や SIGTERM では停止できません。

tty を有効化したコンテナにアタッチした状態（例： -t を使って起動）では、docker attach コマンドを使った標準入力のリダイレクトは禁止されています。

デタッチ・シーケンスの上書き

必要であれば、デタッチ用の Docker キー・シーケンスの設定を上書きできます。Docker デフォルトのキー・シーケンスが他のアプリケーションと重複している場合に役立ちます。デタッチ用キー・シーケンスを指定するには、２つの方法があります。１つはコンテナごとに設定するか、あるいは全体に対してのプロパティを設定します。

個々のコンテナに対するシーケンスを上書きするには、docker attach コマンドに --detach-keys="<シーケンス>" を指定します。 <シーケンス> の書式は、 [a-Z] までの文字を使うか、 ctrl- と次の項目を組み合わせます。

- a-z （小文字のアルファベット文字列）
- @ （アット記号）
- [（左かっこ）
- \\ （２つのバックスラッシュ）
- _ （アンダースコア）
- ^ （キャレット）

例えば、a、ctrl-a、x、ctrl-\\は、いずれも有効なキー・シーケンスです。全てのコンテナに対する異なったキー・シーケンスを設定するには、 設定ファイル のセクションをご覧ください。

例

```
$ docker run -d --name topdemo ubuntu /usr/bin/top -b
$ docker attach topdemo
top - 02:05:52 up  3:05,  0 users,  load average: 0.01, 0.02, 0.05
Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.1%us,  0.2%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:    373572k total,   355560k used,    18012k free,    27872k buffers
Swap:   786428k total,        0k used,   786428k free,   221740k cached

PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1 root      20   0 17200 1116  912 R    0  0.3   0:00.03 top

 top - 02:05:55 up  3:05,  0 users,  load average: 0.01, 0.02, 0.05
 Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
 Cpu(s):  0.0%us,  0.2%sy,  0.0%ni, 99.8%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
 Mem:    373572k total,   355244k used,    18328k free,    27872k buffers
 Swap:   786428k total,        0k used,   786428k free,   221776k cached

   PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
     1 root      20   0 17208 1144  932 R    0  0.3   0:00.03 top


 top - 02:05:58 up  3:06,  0 users,  load average: 0.01, 0.02, 0.05
 Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
 Cpu(s):  0.2%us,  0.3%sy,  0.0%ni, 99.5%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
 Mem:    373572k total,   355780k used,    17792k free,    27880k buffers
 Swap:   786428k total,        0k used,   786428k free,   221776k cached

 PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
     1 root      20   0 17208 1144  932 R    0  0.3   0:00.03 top
^C$
$ echo $?
0
$ docker ps -a | grep topdemo
7998ac8581f9        ubuntu:14.04        "/usr/bin/top -b"   38 seconds ago      Exited (0) 21 seconds
ago                             topdemo
```

次の２つめの例は、docker attach コマンドで処理された終了コードが、bash プロセスに戻ってきても使えることが分かります。

```
$ docker run --name test -d -it debian
275c44472aebd77c926d4527885bb09f2f6db21d878c75f0a1c212c03d3bcfab
$ docker attach test
$$ exit 13
exit
$ echo $?
13
$ docker ps -a | grep test
275c44472aeb        debian:7            "/bin/bash"          26 seconds ago      Exited (13) 17 seconds
ago                             test
```

cp

使い方: docker cp [オプション] コンテナ:パス ローカル・パス|-
docker cp [オプション] ローカル・パス|- コンテナ:パス

コンテナとローカル・ファイルシステム間でファイルとフォルダをコピーする

--help 使い方の表示

概要の１つめは、コンテナのファイルシステム上のパスにある内容をローカルのパスにコピーするために、docker cp が役立ちます。

概要の２つめは、ローカルパス（あるいは STDIN で - で指定した tar アーカイブを流し込み）に含まれる内容を、ローカルマシン上からコンテナのファイルシステム上のパスにコピーします。

コピーはコンテナが実行中でも停止中でも可能です。パスとはファイルまたはディレクトリです。docker cp コマンドは、全てのコンテナ:パス値をコンテナ内の /（ルート）ディレクトリとみなします。つまり、転送先の冒頭にスラッシュを付けるのはオプションです。つまり、コマンド compassionate_darwin:/tmp/foo/myfile.txt と compassionate_darwin:tmp/foo/myfile.txt は同一です。ローカルパスの値は絶対パスではなく、現在の作業ディレクトリからの相対パスとみなされます。

動作は一般的な Unix ユーティリティ cp -a に似ています。可能であればパーミッションと一緒に、ディレクトリも再帰的にコピーします。所有者の権限は、転送終了時にユーザとプライマリ・グループが指定されます。例えば、ファイルをコンテナにコピーすると、UID:GID は root ユーザとして作成されます。ファイルをローカルのマシン上にコピーする時は、docker cp コマンドを実行したユーザの UID:GID が作成されます。もしも docker cp で -L オプションを指定すると、送信元パスのシンボリック・リンクをフォローします。docker cp は送信先パスが存在しなければ、親ディレクトリを作成しません。

パスは / で分離され、１つめの引数は送信元のパス、２つめの引数は送信先のパスとみなされます。次のように動作します。

- 送信元のパスでファイルを指定する
  - 送信先のパスが存在しない
    - 送信先のパスにファイルを作成し、ファイルを保存する
  - 送信先のパスが存在せず、最後に / がある場合
    - エラーが発生：送信先ディレクトリが存在しなくてはいけない
  - 送信先のパスが存在し、ファイルである場合
    - 送信元にあるファイルで、送信先のファイル内容を上書きする
  - 送信先のパスが存在し、ディレクトリの場合
    - 送信元のパスにある名前を元に、対象のディレクトリにファイルをコピー
- 送信元のパスでディレクトリを指定する
  - 送信先のパスが存在しない
    - 送信先のパスにディレクトリを作成し、送信元ディレクトリに含まれる内容をコピーする
  - 送信先のパスが存在し、ファイルである場合
    - エラーが発生：ディレクトリをファイルにコピーできない
  - 送信先のパスが存在し、ディレクトリの場合
    - 送信元のパスが / で終わる場合
      - 送信元ディレクトリを対象のディレクトリにコピー
    - 送信元のパスが / で終わらない場合

送信元ディレクトリの内容を対象のディレクトリにコピー

コマンドで使う**送信元のパス**と**送信先のパス**は上記のルールに従う必要があります。**送信元のパス**がローカル上かつシンボリックリンクの場合は、その対象ではなくシンボリックリンクがコピーされます。

コロン（ **:** ）は**コンテナ**と**パス**のデリミタ（区切り文字）として使われますが、 **:** は `file:name.txt` のように有効な**ローカルのパス**としても使われます。この曖昧さを解決するには、**ローカルパス**で **:** を明確な絶対パス・相対パスの指定に使います。例：

```
`/path/to/file:name.txt` or `./file:name.txt`
```

`/proc`、`/sys`、`/dev` 配下にあるリソースのような特定のシステムファイルはコピーできません。これらはコンテナの中にマウントされます。

送信元のパスに `-` を使うと、`STDIN`（標準入力）を tar アーカイブの内容として流し込みます。このコマンドにより、対象となるコンテナ上にあるファイルシステムの**送信先パス**に展開します。この場合、**送信先パス**にはディレクトリを指定する必要があります。 **送信先パス**に `-` を使うと、tar アーカイブを `STDOUT`（標準出力）します。

ローカルパスの 1 番めの引数に `-` を使うと、tar アーカイブからの内容を `STDIN`（標準入力）としてストリーム（流し込み）ます。これにより、対象となるコンテナのファイルシステムにある**パス**に展開します。元となるコンテナのリソースに含まれる内容が、tar アーカイブとして `STDOUT`（標準出力）にストリーム出力します。

クリエイト
create

新しいコンテナを作成します。

使い方: docker create [オプション] イメージ [コマンド] [引数...]

新しいコンテナを作成

-a, --attach=[]	STDIN、STDOUT、STDERR にアタッチする
--add-host=[]	ホストから IP アドレスのマッピングをカスタマイズして追加（host:ip）
--blkio-weight=0	ブロック IO ウエイト（相対ウエイト）
--blkio-weight-device=[]	ブロック IO ウエイト（相対デバイス・ウエイト。 書式：`デバイス名:ウエイト`）
--cpu-shares=0	CPU 共有（相対ウエイト）
--cap-add=[]	Linux ケーパビリティの追加
--cap-drop=[]	Linux ケーパビリティの削除
--cgroup-parent=""	コンテナ用のオプション親 cgroup を指定
--cidfile=""	コンテナ ID をファイルに書き出し
--cpu-period=0	CPU CFS（Completely Fair Scheduler）ペイロードの制限
--cpu-quota=0	CPU CFS（Completely Fair Scheduler）クォータの制限
--cpuset-cpus=""	実行を許可する CPU（0-3, 0,1）
--cpuset-mems=""	実行を許可するメモリ必要量（0-3, 0,1）
--device=[]	ホスト・デバイスをコンテナに追加
--device-read-bps=[]	デバイスからの読み込みレート（バイト/秒）を制限
--device-read-iops=[]	デバイスからの読み込みレート（IO/秒）を制限
--device-write-bps=[]	デバイスへの書き込みレート（バイト/秒）を制限 （例: --device-write-bps=/dev/sda:1mb）
--device-write-iops=[]	デバイスへの書き込みレート（IO/秒）を制限 （例: --device-write-iops=/dev/sda:1000）
--disable-content-trust=true	イメージの認証をスキップ
--dns=[]	カスタム DNS サーバの指定
--dns-opt=[]	カスタム DNS オプションの指定
--dns-search=[]	カスタム DNS 検索ドメインの指定
-e, --env=[]	環境変数を指定
--entrypoint=""	イメージのデフォルト ENTRYPOINT を上書き
--env-file=[]	ファイルから環境変数を読み込み
--expose=[]	ポートまたはポート範囲を露出
--group-add=[]	参加するグループを追加
-h, --hostname=""	コンテナのホスト名
--help	使い方の表示
-i, --interactive	アタッチしていなくても STDIN を開き続ける
--ip=""	コンテナの IPv4 アドレス（例: 172.30.100.104）
--ip6=""	コンテナの IPv6 アドレス（例: 2001:db8::33）
--ipc=""	使用する IPC 名前空間
--isolation=""	コンテナの分離（独立）技術
--kernel-memory=""	Kernel メモリ上限
-l, --label=[]	コンテナにメタデータを指定（例: --label=com.example.key=value）
--label-file=[]	行ごとにラベルを記述したファイルを読み込み
--link=[]	他のコンテナへのリンクを追加
--log-driver=""	コンテナ用のログ記録ドライバを追加
--log-opt=[]	ログドライバのオプションを指定
-m, --memory=""	メモリ上限
--mac-address=""	コンテナの MAC アドレス（例： 92:d0:c6:0a:29:33）
--memory-reservation=""	メモリのソフト上限
--memory-swap=""	整数値の指定はメモリにスワップ値を追加。-1 は無制限スワップを有効化
--memory-swappiness=""	コンテナ用メモリのスワップ程度を調整。整数値の 0 から 100 で指定

```
  --name=""                    コンテナに名前を割り当て
  --net="bridge"   : コンテナをネットワークに接続
                               'bridge': docker ブリッジ上でコンテナ用に新しいネットワーク・
                               スタックを作成
                               'none': コンテナにネットワーク機能を付けない
                               'container:<name|id>': 他のコンテナ用ネットワーク・スタックを再利用
                               'host': コンテナ内でホスト側ネットワーク・スタックを使用
                               'NETWORK': 「docker network create」コマンドでユーザ作成した
                               ネットワークを使用
  --net-alias=[]               コンテナにネットワーク内部用のエイリアスを追加
  --oom-kill-disable           コンテナの OOM Killer を無効化するかどうか指定
  --oom-score-adj=0            コンテナに対してホスト側の OOM 優先度を設定 ( -1000 〜 1000 を指定)
  -P, --publish-all            全ての露出ポートをランダムならポートに公開
  -p, --publish=[]             コンテナのポートをホスト側に公開
  --pid=""                     使用する PID 名前空間
  --pids-limit=-1               コンテナの pids 制限を調整 (kernel 4.3 以上は -1 で無制限に設定)
  --privileged                 このコンテナに対して拡張権限を与える
  --read-only                  コンテナのルート・ファイルシステムを読み込み専用としてマウント
  --restart="no"               再起動ポリシー (no, on-failure[:max-retry], always, unless-stopped)
  --security-opt=[]            セキュリティ・オプション
  --stop-signal="SIGTERM"      コンテナの停止シグナル
  --shm-size=[]                `/dev/shm` のサイズ。書式は `<数値><単位>`. `数値` は必ず `0` より大き
い。単位はオプションで `b` (bytes)、 `k` (kilobytes)、 `m` (megabytes)、 `g` (gigabytes) を指定可能。
単位を指定しなければ、システムは bytes を使う。数値を指定しなければ、システムは `64m` を使う
  -t, --tty                    疑似ターミナル (pseudo-TTY) を割り当て
  -u, --user=""                ユーザ名または UID
  --userns=""                  コンテナのユーザ名前空間
                               'host': Docker ホストで使うユーザ名前空間
                               '': Docker デーモンのユーザ名前空間を指定するには `--userns-remap`
                                 オプションを使う
  --ulimit=[]                  Ulimit オプション
  --uts=""                     使用する UTS 名前空間
  -v, --volume=[ホスト側ソース:]コンテナ側送信先[:<オプション>]
                               ボリュームを拘束マウント。カンマ区切りで指定
                               `オプション` は [rw|ro], [z|Z], [[r]shared|[r]slave|[r]private],
[nocopy]
                               'ホスト側ソース' は絶対パスまたは名前の値
  --volume-driver=""           コンテナのボリューム・ドライバ
  --volumes-from=[]            指定したコンテナからボリュームをマウント
  -w, --workdir=""             コンテナ内の作業用ディレクトリを指定
```

docker create コマンドは、特定のイメージ上に書き込み可能なコンテナ・レイヤ（層）を作成し、測定のコマンドを実行する準備をします。それから、コンテナ ID を STDOUT（標準出力）に表示します。これは開始したことが無いコンテナに対して docker run -d する時も同様です。それから、docker start <コンテナ ID> コマンドを使って、いつでもコンテナを実行できます。

コンテナが必要な時に直ちに実行できるよう、事前に設定を済ませて使えるように準備したい場合に役立ちます。新しいコンテナ作成時の初期状態を作成しました。

より詳細に関しては run セクションと Docker run リファレンスをご覧ください。

例

```
$ docker create -t -i fedora bash
6d8af538ec541dd581ebc2a24153a28329acb5268abe5ef868c1f1a261221752
$ docker start -a -i 6d8af538ec5
```

```
bash-4.2#
```

バージョン 1.4.0 以降では、docker create の段階で（docker run も同様）コンテナのボリュームが初期化されます。例えば、データ・ボリュームコンテナを create したら、他のコンテナからも利用可能になります。

```
$ docker create -v /data --name data ubuntu
240633dfbb98128fa77473d3d9018f6123b99c454b3251427ae190a7d951ad57
$ docker run --rm --volumes-from data ubuntu ls -la /data
total 8
drwxr-xr-x  2 root root 4096 Dec  5 04:10 .
drwxr-xr-x 48 root root 4096 Dec  5 04:11 ..
```

同様に、 ホスト側のディレクトリをバインドするボリューム・コンテナを作成したら、次に処理するコンテナからも利用可能になります。

```
$ docker create -v /home/docker:/docker --name docker ubuntu
9aa88c08f319cd1e4515c3c46b0de7cc9aa75e878357b1e96f91e2c773029f03
$ docker run --rm --volumes-from docker ubuntu ls -la /docker
total 20
drwxr-sr-x  5 1000 staff  180 Dec  5 04:00 .
drwxr-xr-x 48 root root  4096 Dec  5 04:13 ..
-rw-rw-r--  1 1000 staff 3833 Dec  5 04:01 .ash_history
-rw-r--r--  1 1000 staff  446 Nov 28 11:51 .ashrc
-rw-r--r--  1 1000 staff   25 Dec  5 04:00 .gitconfig
drwxr-sr-x  3 1000 staff   60 Dec  1 03:28 .local
-rw-r--r--  1 1000 staff  920 Nov 28 11:51 .profile
drwx--S---  2 1000 staff  460 Dec  5 00:51 .ssh
drwxr-xr-x 32 1000 staff 1140 Dec  5 04:01 docker
```

ディフ
diff

使い方: docker diff [オプション] コンテナ

コンテナのファイルシステムに対する変更を調査

--help 使い方の表示

コンテナのファイルシステム上で、変更したファイルやディレクトリの一覧を表示します。

- A - 追加（Add）
- D - 削除（Delete）
- C - 変更（Change）

実行例：

```
$ docker diff 7bb0e258aefe

C /dev
A /dev/kmsg
C /etc
A /etc/mtab
A /go
A /go/src
A /go/src/github.com
A /go/src/github.com/docker
A /go/src/github.com/docker/docker
A /go/src/github.com/docker/docker/.git
....
```

イベンツ
events

使い方: docker events [オプション]

サーバからリアルタイムのイベントを取得

```
  -f, --filter=[]    指定した状態に基づき出力をフィルタ
  --help             使い方の表示
  --since=""         タイムスタンプ以降に発生した全てのイベントを表示
  --until=""         タイムスタンプまで発生したイベントを表示
```

Docker コンテナは以下のイベントを報告します。

```
attach, commit, copy, create, destroy, die, exec_create, exec_start, export, kill, oom, pause, rename,
resize, restart, start, stop, top, unpause, update
```

Docker イメージは以下のイベントを報告します。

```
delete, import, pull, push, tag, untag
```

Docker ボリュームは以下のイベントを報告します。

```
create, mount, unmount, destroy
```

Docker ネットワークは以下のイベントを報告します。

```
create, connect, disconnect, destroy
```

--since と --until パラメータでは、Unix タイムスタンプ、RFC 3339 の dates、Go 言語の期間文字列 (例：10m、1h30m) をクライアントのマシン時刻から相対的に扱えます。 --since オプションを指定しなければ、コマンドは新しく追加されたか現在のイベントのみ表示します。

フィルタリング

フィルタリング・フラグ（ -f と --filter）は key=value の形式です。複数のフィルタを使いたい場合は、複数回フラグを指定します（例： --filter "foo=bar" --filter "bif=baz" ）。

同じフィルタを複数回指定したら、「OR」（または）という条件として処理します。例えば --filter container=588a23dac085 --filter container=a8f7720b8c22 は、コンテナ 588a23dac085 かコンテナ a8f7720b8c22 のイベントを表示します。

複数のフィルタを使えば、「AND」（および）という条件として処理します。例えば --filter container=588a23dac085 --filter event=start は、コンテナ 588a23dac085 のイベントタイプが start のイベントのみ表示します。

現時点でサポートされているフィルタは次の通りです。

- コンテナ（ container=<名前か ID> ）
- イベント（ event=<イベント・タイプ> ）
- イメージ（ image=<タグか ID> ）
- ラベル（ label=<key> または label=<key>=<value> ）

例

この例には２つのシェルが必要です。
シェル１：イベント一覧を表示：

```
$ docker events
```

シェル２：コンテナを開始して停止

```
$ docker start 4386fb97867d
$ docker stop 4386fb97867d
$ docker stop 7805c1d35632
```

シェル１：(再度実行したら、イベントを表示)

```
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) start
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) die
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) stop
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8) die
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8) stop
```

時間を指定したら、過去のイベントを表示：

```
$ docker events --since 1378216169
2014-03-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) die
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) stop
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8) die
2014-03-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8) stop

$ docker events --since '2013-09-03'
2014-09-03T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) start
2014-09-03T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) die
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) stop
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8) die
2014-09-03T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8) stop

$ docker events --since '2013-09-03T15:49:29'
2014-09-03T15:49:29.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) die
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) stop
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8) die
2014-09-03T15:49:29.999999999Z07:00 7805c1d35632: (from redis:2.8) stop
```

この例では、過去３分間に発生した全イベントを表示しています。クライアント側のマシン上からの相対的な時間です。

```
$ docker events --since '3m'
2015-05-12T11:51:30.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) die
2015-05-12T15:52:12.999999999Z07:00 4 4386fb97867d: (from ubuntu-1:14.04) stop
2015-05-12T15:53:45.999999999Z07:00  7805c1d35632: (from redis:2.8) die
2015-05-12T15:54:03.999999999Z07:00  7805c1d35632: (from redis:2.8) stop
```

イベントをフィルタします：

```
$ docker events --filter 'event=stop'
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) stop
2014-09-03T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8) stop

$ docker events --filter 'image=ubuntu-1:14.04'
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) start
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) die
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) stop

$ docker events --filter 'container=7805c1d35632'
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8) die
2014-09-03T15:49:29.999999999Z07:00 7805c1d35632: (from redis:2.8) stop

$ docker events --filter 'container=7805c1d35632' --filter 'container=4386fb97867d'
2014-09-03T15:49:29.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) die
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) stop
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8) die
2014-09-03T15:49:29.999999999Z07:00 7805c1d35632: (from redis:2.8) stop

$ docker events --filter 'container=7805c1d35632' --filter 'event=stop'
2014-09-03T15:49:29.999999999Z07:00 7805c1d35632: (from redis:2.8) stop

$ docker events --filter 'container=container_1' --filter 'container=container_2'
2014-09-03T15:49:29.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) die
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14.04) stop
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8) die
2014-09-03T15:49:29.999999999Z07:00 7805c1d35632: (from redis:2.8) stop
```

エクゼク
exec

使い方: docker exec [オプション] コンテナ コマンド [引数...]

実行中のコンテナでコマンドを実行

```
  -d, --detach=false          デタッチド・モード: コマンドをバックグラウンドで実行
  --detach-keys               デタッチド・コンテナに特定のエスケープ・キー・シーケンスを設定
  --help=false                使い方の表示
  -i, --interactive=false     アタッチしていなくても STDIN をオープンにし続ける
  --privileged=false          コマンドに拡張 Linux ケーパビリティの追加
  -t, --tty=false             疑似ターミナル (pseudo-TTY) の割り当て
  -u, --user=                 ユーザ名か UID (書式: <名前|uid>[:<グループ|gid>])
```

docker exec コマンドは実行中のコンテナ内で、新しいコマンドを実行します。

docker exec コマンドが使えるのは、コンテナのプラマリ・プロセス（ PID 1 ）が実行中の時のみです。そして、コンテナが再起動されても、こちらは再起動されません。

コンテナを一時停止すると、docker exec コマンドは停止し、エラーになります。

```
$ docker pause test
test
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
  PORTS               NAMES
1ae3b36715d2        ubuntu:latest       "bash"              17 seconds ago      Up 16 seconds (Paused)
                      test
$ docker exec test ls
FATA[0000] Error response from daemon: Container test is paused, unpause the container before exec
$ echo $?
1
```

例

```
$ docker run --name ubuntu_bash --rm -i -t ubuntu bash
```

これは ubuntu_bash という名前のコンテナを作成し、Bash セッションを開始します。

```
$ docker exec -d ubuntu_bash touch /tmp/execWorks
```

こちらは実行中の ubuntu_bash コンテナ内で、新しいファイル /tmp/execWorks をバックグラウンドで作成します。

```
$ docker exec -it ubuntu_bash bash
```

こちらは ubuntu_bash コンテナ内に新しい Bash セッションを作成します。

kill（キル）

使い方: docker kill [オプション] コンテナ [コンテナ...]

実行中のコンテナを SIGKILL か指定したシグナルで停止

```
--help                  使い方の表示
-s, --signal="KILL"     コンテナに送信するシグナル
```

コンテナの中のメイン・プロセス（ PID 1 ）に対して SIGKILL を送信するか、 --signal オプションで指定したシグナルを送信します。

ENTRYPOINT と CMD をシェル形式で実行している場合は、 /bin/sh -c のサブコマンドとして実行されていますので、シグナルを受け取ることができません。つまり、(シェル形式では) コンテナの PID 1 は Unix シグナルを受け取りません。

ログズ
logs

使い方: docker logs [オプション] コンテナ

コンテナのログを取得

-f, --follow=false	ログの出力をフォロー（表示し続ける）
--help	使い方の表示
--since=""	タイムスタンプ以降のログを表示
-t, --timestamps=false	タイムスタンプを表示
--tail="all"	ログの最後から指定した行数を表示

このコマンドが使えるのは、コンテナが json-file と journald ロギング・ドライバを使う時のみです。

docker logs コマンドは、コンテナの実行時から現在に至るまでのログを、逐次表示します。

docker logs --follow および docker logs -f コマンドは、コンテナの STDOUT と STDERR から新しい出力があれば、表示し続けます。

負の値の指定や、 --tail に値を付けなければ、全てのログを表示します。

docker logs --timestamp コマンドは RFC3339 ナノ秒タイムスタンプ[*1] を追加します。例えば 2014-09-16T06:17:46.000000000Z のように、各ログ行に追加されます。タイムスタンプはナノ秒で表示されるため、不要な場合でもゼロが付加されます。

--since オプションは指定した日時以降のログを表示します。指定できる日付は RFC 3339 date、UNIX タイムスタンプ、Go 言語の期間文字（例：1m30s、3h）です。Docker はクライアント側のマシン上からの相対時間を計算します。 --since オプションは --follow と --tail と同時に使えます。

*1 https://golang.org/pkg/time/#pkg-constants

pause（ポーズ）

使い方: docker pause [オプション] コンテナ [コンテナ...]

コンテナ内の全てのプロセスを一時停止

--help 使い方の表示

docker pause コマンドは cgroup を凍結（freezer）するもので、コンテナ内の全てのプロセスを一時停止（suspend）します。これまで、プロセスの一時停止には SIGSTOP シグナルが使われてきました。これはプロセスが一時停止状態に見えるようにするためのものです。cgroup の凍結により、プロセスの状態は分からなくなり、操作できなくなります。再開されるまで、一時停止状態のままです。

更に詳しい詳細については cgroup freezer ドキュメント[*1] をご覧ください。

*1 https://www.kernel.org/doc/Documentation/cgroups/freezer-subsystem.txt

ポート
port

使い方: docker port [オプション] コンテナ [プライベート・ポート[/プロトコル]]

コンテナに対するマッピング（割り当て）を一覧表示
あるいは NAT されたプライベート・ポートを探して表示

--help 使い方の表示

プライベート・ポートや、特定の割り当て状況を指定しなければ、全てのポートの割り当て状況（マッピング）を確認できます。

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS                                       NAMES
b650456536c7        busybox:latest      top                 54 minutes ago      Up 54 minutes
0.0.0.0:1234->9876/tcp, 0.0.0.0:4321->7890/tcp   test
$ docker port test
7890/tcp -> 0.0.0.0:4321
9876/tcp -> 0.0.0.0:1234
$ docker port test 7890/tcp
0.0.0.0:4321
$ docker port test 7890/udp
2014/06/24 11:53:36 Error: No public port '7890/udp' published for test
$ docker port test 7890
0.0.0.0:4321
```

ps

使い方: docker ps [オプション]

コンテナの一覧

```
  -a, --all             全てのコンテナを表示（デフォルトは実行中コンテナのみ表示）
  -f, --filter=[]       以下の状況に応じて出力をフィルタ:
                         - exited=<整数> 終了コードを <整数> で指定
                         - label=<key> または label=<key>=<value>
                         - status=(created|restarting|running|paused|exited)
                         - name=<文字列> コンテナ名
                         - id=<ID> コンテナ ID
                         - before=(<コンテナ名>|<コンテナ ID>)
                         - since=(<コンテナ名>|<コンテナ ID>)
                         - ancestor=(<イメージ名>[:タグ]|<イメージ ID>|<イメージ@ダイジェスト値>)
                                  - 特定のイメージや子孫から作成されたコンテナ
                         - volume=(<ボリューム名>|<マウント・ポイント>)
  --format=[]           Go テンプレートを使いコンテナの表示を整形
  --help                使い方の表示
  -l, --latest          最後に作成したコンテナを表示（どのような状態でも）
  -n=-1                 直近で作成した n コンテナを表示（どのような状態でも）
  --no-trunc            トランケート（truncate）を出力しない
  -q, --quiet           整数値の ID のみ表示
  -s, --size            合計ファイルサイズを表示
```

docker ps --no-trunc を実行すると、リンクされた 2 つのコンテナを表示します。

```
$ docker ps
CONTAINER ID        IMAGE                     COMMAND              CREATED             STATUS              PORTS               NAMES
4c01db0b339c        ubuntu:12.04              bash                 17 seconds ago      Up 16 seconds       3300-3310/tcp       webapp
d7886598dbe2        crosbymichael/redis:latest   /redis-server --dir   33 minutes ago      Up 33 minutes       6379/tcp            redis,webapp/db
```

デフォルトの docker ps は実行中のコンテナだけ表示します。全てのコンテナを表示するには docker ps -a を使います。

docker ps は、可能であれば公開ポートのグループを範囲で表示します。例えば、コンテナが 100、101、102 を公開している場合、PORT 列に 100-102/tcp と表示します。

フィルタリング

フィルタリング・フラグ（ -f と --filter）の形式は key=value の組です。複数のフィルタを指定するには、複数のフラグを使います（例： --filter "foo=bar" --filter "bif=baz"）。

現在、以下のフィルタをサポートします。

- ID（コンテナの ID）
- ラベル（ label=<key> か label=<key>=<value> ）
- 名前（コンテナの名前）
- 終了コード（整数値 - コンテナの終了コード。実用的なのは --all）
- 先祖/ancestor（ <イメージ名>[:<タグ>] 、 <イメージ ID> 、 <イメージ@digest> ） - 特定のイメージから作られた子コンテナを作成します。

label

label フィルタに一致する条件は、コンテナ自身に割り当てられている label か、その label と値です。
次のフィルタは、color ラベルがどのような値を持っているかに拘わらず、一致するものを表示します。

```
$ docker ps --filter "label=color"
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
673394ef1d4c        busybox             "top"               47 seconds ago      Up 45 seconds
                  nostalgic_shockley
d85756f57265        busybox             "top"               52 seconds ago      Up 51 seconds
                  high_albattani
```

次のフィルタは color ラベルの値が blue に一致するコンテナを表示します。

```
$ docker ps --filter "label=color=blue"
CONTAINER ID        IMAGE               COMMAND             CREATED              STATUS
PORTS               NAMES
d85756f57265        busybox             "top"               About a minute ago   Up About a minute
                   high_albattani
```

name

name フィルタはコンテナ名の全て、または一部に一致するコンテナを表示します。
次のフィルタは nostalgic_stallman 文字列を含む名前のコンテナを表示します。

```
$ docker ps --filter "name=nostalgic_stallman"
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
9b6247364a03        busybox             "top"               2 minutes ago       Up 2 minutes
                  nostalgic_stallman
```

あるいは、一部が一致する場合でも、次のようにフィルタできます。

```
$ docker ps --filter "name=nostalgic"
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
715ebfcee040        busybox             "top"               3 seconds ago       Up 1 seconds
                  i_am_nostalgic
9b6247364a03        busybox             "top"               7 minutes ago       Up 7 minutes
                  nostalgic_stallman
673394ef1d4c        busybox             "top"               38 minutes ago      Up 38 minutes
                  nostalgic_shockley
```

exited

exited は、コンテナの終了コードに一致するものでフィルタします。例えば、正常終了したコンテナでフィルタをするには、次のようにします。

```
$ docker ps -a --filter 'exited=0'
CONTAINER ID        IMAGE             COMMAND               CREATED             STATUS
    PORTS                       NAMES
ea09c3c82f6e        registry:latest   /srv/run.sh           2 weeks ago         Exited (0) 2 weeks ago
  127.0.0.1:5000->5000/tcp   desperate_leakey
106ea823fe4e        fedora:latest     /bin/sh -c 'bash -l'  2 weeks ago         Exited (0) 2 weeks ago
```

```
                                determined_albattani
48ee228c9464        fedora:20           bash                    2 weeks ago         Exited (0) 2 weeks ago
                                  tender_torvalds
```

status

status はコンテナの状態が一致するものでフィルタします。フィルタとして使えるのは created、restarting、running、paused、exited、dead です。例えば、running（実行中）のコンテナでフィルタするには、次のようにします。

```
$ docker ps --filter status=running
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
715ebfcee040        busybox             "top"               16 minutes ago      Up 16 minutes
                    i_am_nostalgic
d5c976d3c462        busybox             "top"               23 minutes ago      Up 23 minutes
                    top
9b6247364a03        busybox             "top"               24 minutes ago      Up 24 minutes
                    nostalgic_stallman
```

paused コンテナでフィルタをするには：

```
$ docker ps --filter status=paused
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
      PORTS               NAMES
673394ef1d4c        busybox             "top"               About an hour ago   Up About an hour (Paused)
                       nostalgic_shockley
```

ancestor

ancestor（先祖）フィルタはコンテナのベースとなったイメージや、その派生に一致するものです。フィルタは以下の形式で指定できます。

```
イメージ
イメージ:タグ
イメージ:タグ@digest
ショート ID
フル ID
```

タグを指定しなければ、latest タグが使われます。例えば、最新（latest）の ubuntu イメージでフィルタするには：

```
$ docker ps --filter ancestor=ubuntu
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
919e1179bdb8        ubuntu-c1           "top"               About a minute ago  Up About a minute
                    admiring_lovelace
5d1e4a540723        ubuntu-c2           "top"               About a minute ago  Up About a minute
                    admiring_sammet
82a598284012        ubuntu              "top"               3 minutes ago       Up 3 minutes
                    sleepy_bose
bab2a34ba363        ubuntu              "top"               3 minutes ago       Up 3 minutes
                    focused_yonath
```

ubuntu-c1 イメージをベースにするコンテナ、この例では ubuntu の子供に一致するものを表示：

```
$ docker ps --filter ancestor=ubuntu-c1
CONTAINER ID        IMAGE               COMMAND             CREATED              STATUS
PORTS               NAMES
919e1179bdb8        ubuntu-c1           "top"               About a minute ago   Up About a minute
                  admiring_lovelace
```

ubuntu バージョン 12.04.5 のイメージをベースとするコンテナをフィルタ：

```
$ docker ps --filter ancestor=ubuntu:12.04.5
CONTAINER ID        IMAGE               COMMAND             CREATED              STATUS
PORTS               NAMES
82a598284012        ubuntu:12.04.5      "top"               3 minutes ago        Up 3 minutes
                  sleepy_bose
```

レイヤ d0e008c6cf02 あるいはイメージをベースにしたコンテナでフィルタします。

```
 $ docker ps --filter ancestor=d0e008c6cf02
CONTAINER ID        IMAGE               COMMAND             CREATED              STATUS
PORTS               NAMES
82a598284012        ubuntu:12.04.5      "top"               3 minutes ago        Up 3 minutes
                  sleepy_bose
```

フォーマット

フォーマットのオプション（ --format）は Go テンプレートを使いコンテナの出力を整形します。
Go テンプレートで置き換え可能な一覧は、次の通りです：

.ID	コンテナ ID
.Image	イメージ ID
.Command	クォートされたコマンド
.CreatedAt	コンテナが作成された時間
.RunnnngFor	コンテナが起動してからの時間
.Ports	公開しているポート
.Status	コンテナのステータス
.Size	コンテナの容量
.Names	コンテナ名
.Lables	コンテナに割り当てられた全ラベル
.Label	コンテナに対する特定のラベル。例： {{.Label "com.docker.swarm.cpu"}}

ps コマンドに --format オプションを使えば、テンプレートで指定したデータを出力するだけでなく、table 命令を使うとカラム（例）ヘッダも同様に表示します。
次の例はヘッダを除くテンプレートを使い、実行中の全てのコンテナに対して、ID と Command エントリを句切って出力します。

```
$ docker ps --format "{{.ID}}: {{.Command}}"
a87ecb4f327c: /bin/sh -c #(nop) MA
01946d9d34d8: /bin/sh -c #(nop) MA
c1d3b0166030: /bin/sh -c yum -y up
41d50ecd2f57: /bin/sh -c #(nop) MA
```

実行中のコンテナのラベルを表形式で出力するには、次のようにします。

```
$ docker ps --format "table {{.ID}}\t{{.Labels}}"
CONTAINER ID        LABELS
a87ecb4f327c        com.docker.swarm.node=ubuntu,com.docker.swarm.storage=ssd
01946d9d34d8
c1d3b0166030        com.docker.swarm.node=debian,com.docker.swarm.cpu=6
41d50ecd2f57        com.docker.swarm.node=fedora,com.docker.swarm.cpu=3,com.docker.swarm.storage=ssd
```

リネーム rename

使い方: docker rename [オプション] 古い名前 新しい名前

コンテナの名前を変更

--help 使い方の表示

docker rename コマンドは、コンテナ名を別の名前にします。

リスタート
restart

使い方: docker restart [オプション] コンテナ [コンテナ...]

コンテナを再起動

```
--help              使い方の表示
-t, --time=10       コンテナを停止するまで待機する秒数
```

rm

使い方: docker rm [オプション] コンテナ [コンテナ...]

１つまたは複数のコンテナを削除

```
  -f, --force            実行中のコンテナを（SIGKILL を使い）強制的に削除
  --help                 使い方の表示
  -l, --link             指定したリンクを削除
  -v, --volumes          コンテナと関連づけられたボリュームを削除
```

例

```
$ docker rm /redis
/redis
```

これは /redis リンクとして参照されているコンテナを削除します。

```
$ docker rm --link /webapp/redis
/webapp/redis
```

これは /webapp/redis でリンクされているコンテナを削除します。

```
$ docker rm --force redis
redis
```

これは /link でリンクされているコンテナを SIGKILL し、コンテナを削除します。

```
$ docker rm $(docker ps -a -q)
```

このコマンドは停止しているコンテナを全て削除します。コマンド docker ps -a -q は終了した全てのコンテナ ID を rm コマンドに渡し、全て削除するものです。実行中のコンテナは削除されません。

```
$ docker rm -v redis
redis
```

このコマンドはコンテナと、コンテナに関連づけられた全ボリュームを削除します。ただし、ボリュームに名前を指定していた場合は、このコマンドでは削除されません。

```
$ docker create -v awesome:/foo -v /bar --name hello redis
hello
$ docker rm -v hello
```

この例では、ボリューム /foo は残り続けますが、ボリューム /bar は削除します。同様に --volumes-from で継承関係にあるボリュームも保持します。

run（ラン）

使い方: docker run [オプション] イメージ [コマンド] [引数...]

新しいコンテナを実行する命令

```
  -a, --attach=[]               STDIN、STDOUT、STDERR にアタッチする
  --add-host=[]                 ホストから IP アドレスのマッピングをカスタマイズして追加（host:ip）
  --blkio-weight=0              ブロック IO ウエイト（相対ウエイト）
  --blkio-weight-device=[]      ブロック IO ウエイト（相対デバイス・ウエイト。書式：`デバイス名:ウエイト`）
  --cpu-shares=0                CPU 共有（相対ウエイト）
  --cap-add=[]                  Linux ケーパビリティの追加
  --cap-drop=[]                 Linux ケーパビリティの削除
  --cgroup-parent=""            コンテナ用のオプション親 cgroup を指定
  --cidfile=""                  コンテナ ID をファイルに書き出し
  --cpu-percent=0               コンテナが実行可能な CPU 使用率のパーセントを制限。Windows のみ
  --cpu-period=0                CPU CFS（Completely Fair Scheduler）ペイロードの制限
  --cpu-quota=0                 CPU CFS（Completely Fair Scheduler）クォータの制限
  --cpuset-cpus=""              実行を許可する CPU（0-3, 0,1）
  --cpuset-mems=""              実行を許可するメモリ必要量（0-3, 0,1）
  -d, --detach                  コンテナをバックグラウンドで実行し、コンテナ ID を表示
  --detach-keys                 コンテナのデタッチに使うエスケープ・キー・シーケンスを設定
  --device=[]                   ホスト・デバイスをコンテナに追加
  --device-read-bps=[]          デバイスからの読み込みレート（バイト/秒）を制限
                                  （例: --device-read-bps=/dev/sda:1mb）
  --device-read-iops=[]         デバイスからの読み込みレート（IO/秒）を制限
                                   （例: --device-read-iops=/dev/sda:1000）
  --device-write-bps=[]         デバイスへの書き込みレート（バイト/秒）を制限
                                   （例: --device-write-bps=/dev/sda:1mb）
  --device-write-iops=[]        デバイスへの書き込みレート（IO/秒）を制限
                                  （例: --device-write-bps=/dev/sda:1000）
  --disable-content-trust=true  イメージの認証をスキップ
  --dns=[]                      カスタム DNS サーバの指定
  --dns-opt=[]                  カスタム DNS オプションの指定
  --dns-search=[]               カスタム DNS 検索ドメインの指定
  -e, --env=[]                  環境変数を指定
  --entrypoint=""               イメージのデフォルト ENTRYPOINT を上書き
  --env-file=[]                 ファイルから環境変数を読み込み
  --expose=[]                   ポートまたはポート範囲を露出
  --group-add=[]                参加するグループを追加
  -h, --hostname=""             コンテナのホスト名
  --help                        使い方の表示
  -i, --interactive             アタッチしていなくても STDIN を開き続ける
  --ip=""                       コンテナの IPv4 アドレス（例: 172.30.100.104）
  --ip6=""                      コンテナの IPv6 アドレス（例: 2001:db8::33）
  --ipc=""                      使用する IPC 名前空間
  --isolation=""                コンテナの分離（独立）技術
  --kernel-memory=""            Kernel メモリ上限
  -l, --label=[]                コンテナにメタデータを指定（例: --label=com.example.key=value）
  --label-file=[]               行ごとにラベルを記述したファイルを読み込み
  --link=[]                     他のコンテナへのリンクを追加
  --log-driver=""               コンテナ用のログ記録ドライバを追加
  --log-opt=[]                  ログドライバのオプションを指定
  -m, --memory=""               メモリ上限
  --mac-address=""              コンテナの MAC アドレス（例： 92:d0:c6:0a:29:33）
```

```
  --io-maxbandwidth=""          システム・デバイスの IO 帯域に対する上限を指定（Windows のみ）。
                                書式は `<数値><単位>`。単位はオプションで `b`（バイト/秒）、
                                `k`（キロバイト/秒）、 `m`（メガバイト/秒）、 `g`（ギガバイト/秒）。
                                単位を指定しなければ、システムはバイト/秒とみなす。
                                --io-maxbandwidth と --io-maxiops は相互排他オプション
  --io-maxiops=0                システム・ドライブの最大 IO/秒に対する上限を指定 *Windows のみ）
                                --io-maxbandwidth と --io-maxiops は相互排他オプション
  --memory-reservation=""       メモリのソフト上限
  --memory-swap=""              整数値の指定はメモリにスワップ値を追加。-1 は無制限スワップを有効化
  --memory-swappiness=""        コンテナ用メモリのスワップ程度を調整。整数値の 0 から 100 で指定
  --name=""                     コンテナに名前を割り当て
  --net="bridge"   : コンテナをネットワークに接続
                                'bridge': docker ブリッジ上でコンテナ用に新ネットワーク・スタック作成
                                'none': コンテナにネットワーク機能を付けない
                                'container:<name|id>': 他のコンテナ用ネットワーク・スタックを再利用
                                'host': コンテナ内でホスト側ネットワーク・スタックを使用
                                'NETWORK': docker network create でユーザが作成したネットワークを使用
  --net-alias=[]                コンテナにネットワーク内部用のエイリアスを追加
  --oom-kill-disable            コンテナの OOM Killer を無効化するかどうか指定
  --oom-score-adj=0             コンテナに対してホスト側の OOM 優先度を設定（ -1000 ～ 1000 を指定）
  -P, --publish-all             全ての露出ポートをランダムならポートに公開
  -p, --publish=[]              コンテナのポートをホスト側に公開
  --pid=""                      使用する PID 名前空間
  --pids-limit=-1                コンテナの pids 制限を調整（kernel 4.3 以上は -1 で無制限に設定）
  --privileged                  このコンテナに対して拡張権限を与える
  --read-only                   コンテナのルート・ファイルシステムを読み込み専用としてマウント
  --restart="no"                再起動ポリシー（no, on-failure[:max-retry], always, unless-stopped）
  --rm                          コンテナ終了時、自動的に削除
  --shm-size=[]                 `/dev/shm` のサイズ。書式は `<数値><単位>`. `数値` は必ず `0` より大き
い。単位はオプションで `b`（bytes）、 `k`（kilobytes）、 `m`（megabytes）、 `g`（gigabytes）を指定可能。
単位を指定しなければ、システムは bytes を使う。数値を指定しなければ、システムは `64m` を使う
  --security-opt=[]             セキュリティ・オプション
  --sig-proxy=true              受信したシグナルをプロセスにプロキシ
  --stop-signal="SIGTERM"       コンテナの停止シグナル
  --storage-opt=[]              コンテナごとにストレージ・ドライバのオプションを指定
  --sysctl[=*[]*]]              実行時に名前空間カーネル・パラメータを調整
  -t, --tty                     疑似ターミナル（pseudo-TTY）を割り当て
  -u, --user=""                 ユーザ名または UID
  --userns=""                   コンテナのユーザ名前空間
                                'host': Docker ホストで使うユーザ名前空間
                                '': Docker デーモンのユーザ名前空間を指定するには
                                   `--userns-remap` オプションを使う
  --ulimit=[]                   Ulimit オプション
  --uts=""                      使用する UTS 名前空間
  -v, --volume=[ホスト側ソース:]コンテナ側送信先[:<オプション>]
                                ボリュームを拘束マウント。カンマ区切りで指定。`オプション` は [rw|ro],
                                [z|Z], [[r]shared|[r]slave|[r]private], [nocopy]
                                'ホスト側ソース' は絶対パスまたは名前の値
  --volume-driver=""            コンテナのボリューム・ドライバ
  --volumes-from=[]             指定したコンテナからボリュームをマウント
  -w, --workdir=""              コンテナ内の作業用ディレクトリを指定
```

docker run コマンドは、まず指定されたイメージ上に書き込み可能なコンテナ・レイヤを create（作成）します。それから、指定されたコマンドを使って start（開始）します。この docker run は、API の/containers/create の後で/containers/(id)/start を実行するのと同じです。以前に使っていたコンテナは docker start で再起動で

きます。全てのコンテナを表示するには docker ps -a を使います。

docker run コマンドは、コンテナの内容を確定するために、docker commit コマンドと連携して使えます。

コンテナをネットワークで接続する詳細については、Docker ネットワーク概要のセクションをご覧ください。

例

名前と疑似 TTY の割り当て（--name、-it）

```
$ docker run --name test -it debian
root@d6c0fe130dba:/# exit 13
$ echo $?
13
$ docker ps -a | grep test
d6c0fe130dba        debian:7            "/bin/bash"          26 seconds ago       Exited (13) 17 seconds
ago                            test
```

この例は debian:latest イメージを使い、test という名称のコンテナを実行します。-it は疑似 TTY（pseudo-TTY）をコンテナの標準入力に接続するよう、Docker に対して命令します。つまり、コンテナ内でインタラクティブな bash シェルを作成します。例の中で、bash シェルを終了コード 13 で終了しています。この終了コードは docker run を呼び出したもの（docker）にも送られ、test コンテナのメタデータに記録されます。

コンテナ ID の取得（--cidfile）

```
$ docker run --cidfile /tmp/docker_test.cid ubuntu echo "test"
```

これはコンテナを作成し、コンソール上に test を表示します。cidfile フラグは Docker に新しいファイルを作成させ、そこにコンテナ ID　を書かせるものです。もしファイルが既に存在している場合、Docker はエラーを返します。docker run を終了したら、Docker はこのファイルを閉じます。

コンテナのケーパビリティ（--privileged）

```
$ docker run -t -i --rm ubuntu bash
root@bc338942ef20:/# mount -t tmpfs none /mnt
mount: permission denied
```

これは**動作しません**。デフォルトでは、カーネルに対して潜在的に危険になりうる処理を破棄します。これには cap_sys_admin も含まれます（ファイルシステムのマウントに必要なものです）。しかしながら、--privileged フラグがあれば、実行できるようになります。

```
$ docker run -t -i --privileged ubuntu bash
root@50e3f57e16e6:/# mount -t tmpfs none /mnt
root@50e3f57e16e6:/# df -h
Filesystem      Size  Used Avail Use% Mounted on
none            1.9G     0  1.9G   0% /mnt
```

--privileged フラグはコンテナに対して**全ての**能力を与えます。また、そのために device cgroup コントローラの制限を昇格します。言い換えますと、コンテナはホスト上であらゆる処理が可能となります。このフラグが存在する時、 Docker の中で Docker を動かすといった特別な使い方ができます。

作業ディレクトリを指定（-w）

```
$ docker  run -w /path/to/dir/ -i -t  ubuntu pwd
```

-w は、指定したディレクトリの中でコマンドを実行します。この例では /path/to/dir で実行します。コンテナ内にパスが存在しなければ、作成されます。

コンテナごとにストレージ・オプションを指定

```
$ docker create -it --storage-opt size=120G fedora /bin/bash
```

これ（容量）はコンテナの作成時にルート・ファイルシステムの容量を 120GB に指定しています。ただし、デフォルトの BaseFS 容量よりも小さく指定できません。

tmpfs のマウント（--tmpfs）

```
$ docker run -d --tmpfs /run:rw,noexec,nosuid,size=65536k my_image
```

--tmpfs フラグはコンテナに対して空の tmfps をマウントします。この時、オプション rw、noexec、nosuid、size=65536k オプションを指定しています。

ボリュームのマウント（-v, --read-only）

```
$ docker  run  -v `pwd`:`pwd` -w `pwd` -i -t  ubuntu pwd
```

-v フラグは現在の作業ディレクトリをコンテナ内にマウントします。 -w によって、コマンドは現在の作業用ディレクトリの中で実行されます。ディレクトリとは、pwd を実行して得られるディレクトリが該当します。このコマンドを組み合わせてコンテナを実行しても、現在の作業ディレクトリの中で実行されるのです。

```
$ docker run -v /doesnt/exist:/foo -w /foo -i -t ubuntu bash
```

ボリュームとしてマウントするホスト側のディレクトリが存在しなければ、Docker は自動的にホスト上にディレクトリを作成します。先ほどの例では、Docker はコンテナ起動前に /doesnt/exit ディレクトリを作成します。

```
$ docker run --read-only -v /icanwrite busybox touch /icanwrite here
```

ボリュームに --read-only を指定して使うことで、コンテナの書き込み可能なファイルを制御できます。--read-only フラグはコンテナのルート・ファイルシステムを読み込み専用としてマウントし、コンテナで指定したボリューム以外での書き込みを禁止します。

```
$ docker run -t -i -v /var/run/docker.sock:/var/run/docker.sock \
  -v /path/to/static-docker-binary:/usr/bin/docker busybox sh
```

Docker Unix ソケットと docker バイナリ（https://get.docker.com から入手）に対するマウントにより、コンテナはホスト側の Docker デーモンに対して作成や各種操作といった完全アクセスをもたらします。

ポートの公開と露出（-p、--expose）

```
$ docker run -p 127.0.0.1:80:8080 ubuntu bash
```

コンテナのポート 8080 を 127.0.0.1 上のポート 80 にバインド（割り当て）します。 Docker ユーザ・ガイドで Docker がどのようにポートを操作するか詳細を説明しています。

```
$ docker run --expose 80 ubuntu bash
```

コンテナのポート 80 を露出（expose）しますが、ホストシステム側のインターフェースには公開しません。

環境変数の設定（-e、--env、--env-file）

```
$ docker run -e MYVAR1 --env MYVAR2=foo --env-file ./env.list ubuntu bash
```

これはコンテナ内におけるシンプルな（配列ではない）環境変数を設定します。この３つのフラグについて説明します。 -e と --env は環境変数と値を指定する場所です。あるいは、もし = が指定されなければ、現在の環境変数がそのまま送られます（例： ホスト上の$MYVAR1 がコンテナ内の$MYVAR1 にセットされます ）。 = が指定されず、クライアント側の環境変数が無い場合は、コンテナ内の環境変数からは削除されます。この３つのフラグ -e、--env、--env-file は何度でも指定できます。

これらの３つのフラグに関係なく、 --env-file が始めに処理され、その後 -e と --env フラグが処理されます。この方法は、必要な時に -e と --env で変数を上書きするために使えます。

```
$ cat ./env.list
TEST_FOO=BAR
$ docker run --env TEST_FOO="This is a test" --env-file ./env.list busybox env | grep TEST_FOO
TEST_FOO=This is a test
```

--env-file フラグは、ファイル名を引数として使います。ファイルの内容は、それぞれの行が VAR=VAL の形式であり、 --env のようなものです。コメント行は、行頭に # を付けます。

```
$ cat ./env.list
TEST_FOO=BAR

# ここはコメント
TEST_APP_DEST_HOST=10.10.0.127
TEST_APP_DEST_PORT=8888
_TEST_BAR=FOO
TEST_APP_42=magic
helloWorld=true
123qwe=bar
org.spring.config=something

# 実行者は環境変数を渡す
TEST_PASSTHROUGH
$ TEST_PASSTHROUGH=howdy docker run --env-file ./env.list busybox env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=5198e0745561
TEST_FOO=BAR
TEST_APP_DEST_HOST=10.10.0.127
TEST_APP_DEST_PORT=8888
_TEST_BAR=FOO
TEST_APP_42=magic
helloWorld=true
TEST_PASSTHROUGH=howdy
HOME=/root
123qwe=bar
org.spring.config=something

$ docker run --env-file ./env.list busybox env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=5198e0745561
TEST_FOO=BAR
```

```
TEST_APP_DEST_HOST=10.10.0.127
TEST_APP_DEST_PORT=8888
_TEST_BAR=FOO
TEST_APP_42=magic
helloWorld=true
TEST_PASSTHROUGH=
HOME=/root
123qwe=bar
org.spring.config=something
```

メタデータをコンテナに設定（-l、--label、--label-file）

ラベルとは key=value のペアであり、コンテナにメタデータを提供します。コンテナに２つのラベルをラベル付けします：

```
$ docker run -l my-label --label com.example.foo=bar ubuntu bash
```

my-label キーが値を指定しなければ、対象のラベルは空の文字列（ "" ）がデフォルトで割り当てられます。複数のラベルを追加するには、ラベルのフラグ（ -l か --label）を繰り返します。

key=value はラベル値を上書きしないよう、ユニークにする必要があります。ラベルが値の違う特定のキーを指定した場合は、以前の値が新しい値に上書きされます。Docker は最新の key=value 指定を使います。

--label-file フラグはファイルから複数のラベルを読み込みます。ラベルとしての区切りは各行の EOL マークが現れるまでです。

```
$ docker run --label-file ./labels ubuntu bash
```

label-file の書式は、環境変数の読み込み書式と似ています（環境変数との違いは、ラベルはコンテナ内で実行中のプロセスから見えません）。以下は label-file 形式の記述例です。

```
com.example.label1="a label"

# これはコメントです
com.example.label2=another\ label
com.example.label3
```

複数のラベル用ファイルを読み込むには、複数回 --label-file フラグを使います。

ラベルの動作に関する詳しい情報は、Docker ユーザ・ガイドの「Label - Docker でカスタム・メタデータを使う」セクションをご覧ください。

コンテナをネットワークに接続（--net）

コンテナ実行時に --net フラグを付けるとネットワークに接続します。次の例は busybox コンテナに my-net ネットワークを追加します。

```
$ docker run -itd --net=my-net busybox
```

また、ユーザ定義ネットワーク上でコンテナを起動時、 --ip と --ipv6 フラグを使い、コンテナに対して IP アドレスを割り当て可能です。

```
$ docker run -itd --net=my-net --ip=10.10.9.75 busybox
```

実行中のコンテナに対してネットワークを追加する時は、docker network connect サブコマンドを使います。

同じネットワークに複数のコンテナを接続できます。接続したら、コンテナは別のコンテナの IP アドレスや名前で簡単に通信できるようになります。overlay ネットワークやカスタム・プラグインは複数のホストへの接続をサポートしています。異なった Docker エンジンが起動していても、コンテナが同じマルチホスト・ネットワーク上であれば、相互に通信できます。

サービス・ディスカバリはデフォルトの bridge ネットワークで利用できません。そのため、デフォルトでは、コンテナは IP アドレスで通信します。コンテナ名で通信するには、リンクされている必要があります。

ネットワークからコンテナを切断するには、docker network disconnect コマンドを使います。

コンテナからボリュームをマウント（--volumes-from）

```
$ docker run --volumes-from 777f7dc92da7 --volumes-from ba8c0c54f0f2:ro -i -t ubuntu pwd
```

--volumes-from フラグは、参照するコンテナで定義されたボリュームをマウントできます。コンテナは --volumes-from 引数を何度も指定できます。コンテナ ID はオプションで末尾に :ro か :rw を指定し、読み込み専用か読み書き可能なモードを個々に指定できます。デフォルトでは、ボリュームは参照しているコンテナと同じモード（読み書き可能か読み込み専用）です。

SELinux のようなラベリング・システムは、コンテナ内にボリューム内容をマウントするにあたり、適切なラベルを必要とします。ラベルが無ければ、対象の領域を使ったコンテナの中では、セキュリティ・システムがプロセスの実行を阻止します。デフォルトでは、Docker は OS によってセットされるラベルを変更しません。

コンテナ内にあるラベルを変更するには、ボリュームのマウントに :z か :Z の２つを末尾に追加できます。これらのサフィックスは、Docker に対して共有ボリューム上のファイル・オブジェクトに対して再度ラベル付けするように伝えます。その結果、Docker は共有コンテントのラベルを使ってラベル付けします。共有ボリュームのラベルは、全てのコンテナを読み書き可能なコンテントにします。 Z オプションは Docker に対してプライベートな共有されないラベルであると伝えます。現在のコンテナのみ、プライベート・ボリュームが使えます。

STDIN・STDOUT・STDERR のアタッチ（-a）

-a フラグは docker run 時にコンテナの STDIN、STDOUT、STDERR をバインドします。これにより、必要に応じて入出力を操作できるようにします。

```
$ echo "test" | docker run -i -a stdin ubuntu cat -
```

これはコンテナの中にデータをパイプし、コンテナ ID をコンテナの STDIN にアタッチして表示します。

```
$ docker run -a stderr ubuntu echo test
```

これはエラーでない限り、何も表示しません。これはコンテナの STDIRR にしかアタッチしていないためです。コンテナのログに STDERR と STDOUT が書き込まれます。

```
$ cat somefile | docker run -i -a stdin mybuilder dobuild
```

これはファイルの内容をコンテナにパイプし、構築するものです。構築が完了するとコンテナ ID が表示され、構築ログは docker logs で取得できます。これはファイルや何かをコンテナ内にパイプし、コンテナで処理が終わるとコンテナ ID を表示するので便利です。

ホスト・デバイスをコンテナに追加（--device）

```
$ docker run --device=/dev/sdc:/dev/xvdc --device=/dev/sdd --device=/dev/zero:/dev/nulo -i -t ubuntu ls
```

```
-l /dev/{xvdc,sdd,nulo}
brw-rw---- 1 root disk 8, 2 Feb  9 16:05 /dev/xvdc
brw-rw---- 1 root disk 8, 3 Feb  9 16:05 /dev/sdd
crw-rw-rw- 1 root root 1, 5 Feb  9 16:05 /dev/nulo
```

デバイスをコンテナに直接さらす必要が度々あります。 --device オプションは、これを可能にします。例えば、特定のブロック・ストレージ・デバイス、ループ・デバイス、オーディオ・デバイスを使うにあたり、コンテナに特権を与えなくても（--privileged フラグを使わずに ）追加でき、アプリケーションが直接使えるようになります。

デフォルトでは、コンテナは read、write、mknod を各デバイスに指定できます。各 --device フラグのオプション設定で、３つの :rwm を利用できます。

```
$ docker run --device=/dev/sda:/dev/xvdc --rm -it ubuntu fdisk  /dev/xvdc

Command (m for help): q
$ docker run --device=/dev/sda:/dev/xvdc:r --rm -it ubuntu fdisk  /dev/xvdc
You will not be able to write the partition table.

Command (m for help): q

$ docker run --device=/dev/sda:/dev/xvdc:rw -it ubuntu fdisk  /dev/xvdc

Command (m for help): q

$ docker run --device=/dev/sda:/dev/xvdc:m --rm -it ubuntu fdisk  /dev/xvdc
fdisk: unable to open /dev/xvdc: Operation not permitted
```

--device はエフェメラルな（短命な）デバイスでは使うべきではありません。信頼できないコンテナが --device を追加しようとしても、ブロック・デバイスは除外されるでしょう。

再起動ポリシー

Docker の --restart はコンテナの再起動ポリシーを指定します。再起動ポリシーは、コンテナの終了後、Docker デーモンが再起動するかどうかを管理します。Docker は次の再起動ポリシーをサポートしています。

ポリシー	説明
no（なし）	コンテナが終了しても、自動的には再起動しません。これがデフォルトです。
on-failure[:最大リトライ数]	コンテナが 0 以外の終了コードで停止したら再起動します。オプションで Docker デーモンが何度再起動を試みるかを指定できます。
always（常に）	コンテナの終了コードに拘わらず、常にコンテナの再起動を試みます。Docker デーモンは無制限に再起動を試みます。また、デーモンの起動時にも、コンテナの状況に拘わらず常に起動を試みます。
unless-stopped（停止していない場合）	コンテナの終了コードに拘わらず、常にコンテナの再起動を試みます。しかし、直近のコンテナが停止状態であったのであれば、デーモンの起動時にコンテナを開始しません。

```
$ docker run --restart=always redis
```

これは redis コンテナを再起動ポリシー always で起動するものです。つまり、コンテナが終了したら Docker がコンテナを再起動します。

再起動ポリシーに関するより詳しい情報は、「Docker run リファレンス」の「再起動ポリシー（--restart)」をご覧ください。

コンテナの hosts ファイルにエントリ追加（--add-host）

--add-host フラグを使い、コンテナの /etc/hosts ファイルに 1 つもしくは複数のホストを追加できます。次の例はホスト名 docker に静的なアドレスを追加しています。

```
$ docker run --add-host=docker:10.180.0.1 --rm -it debian
$$ ping docker
PING docker (10.180.0.1): 48 data bytes
56 bytes from 10.180.0.1: icmp_seq=0 ttl=254 time=7.600 ms
56 bytes from 10.180.0.1: icmp_seq=1 ttl=254 time=30.705 ms
^C--- docker ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 7.600/19.152/30.705/11.553 ms
```

時々、コンテナ内から Docker ホストに対して接続する必要があります。接続のためには、--add-host フラグをコンテナに使い、Docker ホストの IP アドレスを与えます。ホスト側の IP アドレスを確認するには、ip addr show コマンドを使います。

コンテナが何の IPv4 ないし IPv6 ネットワークを使っているかは、ip addr show の結果次第です。次のフラグは、ネットワーク・デバイス eth0 の IPv4 アドレスを指定します。

```
$ HOSTIP=`ip -4 addr show scope global dev eth0 | grep inet | awk '{print \$2}' | cut -d / -f 1`
$ docker run  --add-host=docker:${HOSTIP} --rm -it debian
```

IPv6 は -4 フラグの替わりに -6 を指定します。他のネットワーク・デバイスの場合は eth0 を適切なデバイス名に置き換えます（例えば docker0 ブリッジ・デバイス)。

コンテナ内の ulimits を指定（--ulimit）

コンテナ内で ulimit 設定をするには追加特権が必要であり、デフォルトのコンテナでは指定できません。そこで --ulimit フラグを指定できます。--ulimit はソフト・リミットとハード・リミットを指定できます。<type>=<ソフト・リミット>[:<ハード・リミット>] の形式です。例：

```
$ docker run --ulimit nofile=1024:1024 --rm debian sh -c "ulimit -n"
1024
```

ハード・リミットを指定しなければ、ソフト・リミットが両方の値として使われます。ulimits を設定しなければ、デーモンのデフォルト ulimits を継承します。as オプションは無効化されました。言い換えますと、次のようなスクリプトはサポートされていません：
$ docker run -it --ulimit as=1024 fedora /bin/bash

設定したら適切な syscall が送信されます。Docker は転送に何ら介在しません。値が設定された時のみ処理されます。

nproc を使うには

ulimit フラグに nproc を設定する時とは、nproc で Linux 利用者が利用可能な最大プロセス数をセットするものであり、コンテナに対してではないので注意してください。次の例は、daemon ユーザとして 4 つのコンテナを起動します。

```
docker run -d -u daemon --ulimit nproc=3 busybox top
docker run -d -u daemon --ulimit nproc=3 busybox top
docker run -d -u daemon --ulimit nproc=3 busybox top
docker run -d -u daemon --ulimit nproc=3 busybox top
```

4番めのコンテナは失敗し、“[8] System error: resource temporarily unavailable ” エラーを表示します。これが失敗するのは、実行時に nproc=3 を指定したからです。3つのコンテナが起動したら、daemon ユーザに指定されたプロセスの上限（quota）に達してしまうからです。

コンテナをシグナルで停止

--stop-signal フラグは、システムコールのシグナルを設定します。これは、コンテナを終了する時に送るものです。このシグナルはカーネルの syscall テーブルにある適切な数値と一致する必要があります。例えば 9 や、SIGNAME のような形式のシグナル名（例：SIGKILL）です。

コンテナの分離（独立）技術を指定（--isolation）

このオプションは Docker コンテナを Microsoft Windows 上で使う状況で便利です。 --isolation <値> オプションでコンテナの分離（isolation）技術を指定します。 Linux 上では Linux 名前空間（namespaces）を使う default しかサポートしていません。Linux 上では次の2つのコマンドが同等です。

```
$ docker run -d busybox top
$ docker run -d --isolation default busybox top
```

特に Microsoft Windows 上で daemon オプションを指定していなければ、次の2つのコマンドは同等です。

```
$ docker run -d --isolation default busybox top
$ docker run -d --isolation process busybox top
```

Docker daemon 上で --exec-opt isolation=hyperv オプションを指定したら、各コマンドの実行に hyperv 分離を使った結果を表示します。

```
$ docker run -d --isolation default busybox top
$ docker run -d --isolation hyperv busybox top
```

実行時に名前空間のカーネル・パラメータ（sysctl）を設定

--sysctl はコンテナ内の名前空間におけるカーネル・パラメータ（sysctl）を設定します。例えば、コンテナのネットワーク名前空間で IP 転送を有効にするには、次のようにコマンドを実行します。

```
$ docker run --sysctl net.ipv4.ip_forward=1 someimage
```

全ての sysctl が名前空間で使えるわけではありません。Docker はコンテナ内の sysctl の変更をサポートしません。つまり、コンテナ内だけでなくホスト側も変更します。カーネルが改良されれば、更に多くの sysctl を名前空間内で利用可能になると考えています。

サポート中の sysctl

IPC 名前空間：

kernel.msgmax、 kernel.msgmnb、 kernel.msgmni、 kernel.sem、 kernel.shmall、 kernel.shmmax、 kernel.shmmni、 kernel.shm_rmid_forced、 fs.mqueue.* で始まる sysctl 。

--ipc=host オプションを使う場合は、これら sysctl のオプション指定が許可されません。

ネットワーク名前空間： net.* で始まる sysctl

`--ipc=host` オプションを使う場合は、これら `sysctl` のオプション指定が許可されません。

スタート
start

使い方: docker start [オプション] コンテナ [コンテナ...]

1つまたは複数のコンテナを起動する

-a, --attach	STDIN、STDOUT、STDERR にアタッチする
--detach-keys	コンテナのデタッチに使うエスケープ・キー・シーケンスを設定
--help	使い方の表示
-i, --interactive	コンテナの STDIN にアタッチ

stats（スタッツ）

使い方: docker stats [オプション] [コンテナ...]

１つまたは複数のコンテナのリソース使用状況をライブで流し続ける

```
-a, --all          全てのコンテナを表示（デフォルトは実行中のものだけ）
--help             使い方の表示
--no-stream        ストリームを無効化し、初回の結果しか表示しない
```

docker stats コマンドは実行中のコンテナからライブ・データ・ストリームを返します。特定コンテナの情報のみを取得するには、コンテナ名またはコンテナ ID をスペース区切りで追加します。ここでは停止しているコンテナも指定できますが、停止中のコンテナは何も返しません。

コンテナのリソース使用詳細を知りたい場合は、 /containers/(id)/stats API エンドポイントを使います。

例

docker stats を複数のコンテナに実行します。

```
$ docker stats
CONTAINER           CPU %               MEM USAGE / LIMIT     MEM %               NET I/O
BLOCK I/O
1285939c1fd3        0.07%               796 KiB / 64 MiB        1.21%               788 B / 648 B
3.568 MB / 512 KB
9c76f7834ae2        0.07%               2.746 MiB / 64 MiB      4.29%               1.266 KB / 648 B
12.4 MB / 0 B
d1ea048f04e4        0.03%               4.583 MiB / 64 MiB      6.30%               2.854 KB / 648 B
27.7 MB / 0 B
```

docker stats で複数のコンテナ名・ID を指定します。

```
$ docker stats fervent_panini 5acfcb1b4fd1
CONTAINER           CPU %               MEM USAGE/LIMIT     MEM %               NET I/O
5acfcb1b4fd1        0.00%               115.2 MiB/1.045 GiB   11.03%              1.422 kB/648 B
fervent_panini      0.02%               11.08 MiB/1.045 GiB   1.06%               648 B/648 B
```

ストップ
stop

使い方: docker stop [オプション] コンテナ [コンテナ...]

コンテナに SIGTERM を送信し、一定期間後に SIGKILL を送信

```
  --help              使い方の表示
  -t, --time=10       kill で停止する前に待機する秒数
```

コンテナ内のメイン・プロセスは SIGTERM を受信します。一定期間経過すると、SIGKILL を送ります。

トップ
top

使い方: docker top [オプション] コンテナ [ps オプション]

コンテナで実行中のプロセスを表示

--help 使い方の表示

unpause（アンポーズ）

使い方: docker unpause [オプション] コンテナ [コンテナ...]

コンテナ内の全てのプロセスに対し、一時停止を解除

--help 使い方の表示

docker unpause コマンドは、cgroup freezer を使ってコンテナ内で一時停止している全プロセスを再開します。より詳細については cgroups freezer ドキュメント[*1] をご覧ください。

*1 https://www.kernel.org/doc/Documentation/cgroups/freezer-subsystem.txt

ウエイト
wait

使い方: docker wait [オプション] コンテナ [コンテナ...]

コンテナが停止するのを阻止し、終了コードを表示

--help 使い方の表示

A2.6 Hub・レジストリ用コマンド

login（ログイン）

使い方: docker login [オプション] [サーバ]

Docker レジストリ・サーバに登録またはログインする
サーバの指定が無ければデフォルトで "https://index.docker.io/v1/" を使用

```
-e, --email=""      メールアドレス
--help              使い方の表示
-p, --password=""   パスワード
-u, --username=""   ユーザ名
```

自分でホストしているレジストリにログインするには、サーバ名を指定します。
例：

```
$ docker login localhost:8080
```

docker login の実行には sudo か root になる必要があります。ただし次の場合は除外します。

1. docker-machine を使い、docker engine を自動設定したようなリモート・デーモンに接続時。
2. docker グループに追加されたユーザ。システム上のセキュリティ・リスクになります。docker グループは root と同等のためです。詳細は「Docker デーモンが直面する攻撃」のセクションをご覧ください。

証明書（credential）があれば、あらゆるパブリックないしプライベートなリポジトリにログインできます。ログインしたら、コマンドは符号化(エンコード)した証明書を Linux であれば $HOME/.docker/config.json に、Windows であれば %USERPROFILE%/.docker/config.json に保管します。

sudo docker login を実行したら、証明書は /root/.docker/config.json に保管されます。

証明書ストア

Docker Engine はユーザの証明書（credential）を外部の証明書ストアに保存できます。外部の証明書ストアとは、オペレーティング・システムのネイティブなキーチェーン（keychain）です。Docker 設定ファイルに証明書を保管するよりも、外部のストアを使う方がセキュリティは高いです。

証明書ストアを使うには、キーチェーンや外部ストアと接続する外部のヘルパー・プログラムが必要です。Docker はクライアント・ホスト上の $PATH にヘルパー・プログラムが必要です。

こちらは現時点で利用可能な証明書ヘルパー・プログラムと、ダウンロード先の一覧です。

- D-Bus シークレット・サービス： https://github.com/docker/docker-credential-helpers/releases
- Apple OS X キーチェーン：https://github.com/docker/docker-credential-helpers/releases
- Microsoft Windows 資格情報マネージャ：https://github.com/docker/docker-credential-helpers/releases

使い方

証明書ストアを使うには $HOME/.docker/config.json で Docker Engine に対して指定する必要があります。

```
{
     "credsStore": "osxkeychain"
}
```

既にログイン状態であれば、docker logout を実行し、ファイルから認証情報を削除します。それから docker login を再び実行します。

プロトコル

証明書ヘルパーは、どのようなプログラムやスクリプトでも扱える非常にシンプルなプロトコルです。このプロトコルは Git のアイディアに強く影響を受けていますが、情報を共有する仕組みは違います。

ヘルパーはコマンドのアクションを決めるため、常に 1 番めの引数を使います。ここで利用可能な引数とは store get erase のいずれかです。

store 命令は標準入力の JSON ペイロードを取得します。ペイロードではサーバのアドレス、証明書の指定、ユーザ名、パスワードあるいは識別用トークンを渡します。

```
{
     "ServerURL": "https://index.docker.io/v1",
     "Username": "david",
     "Secret": "passw0rd1"
 }
```

もしシークレット（訳者注：証明書やトークンなどの秘密情報の意味）が識別用トークンを保管する場合、ユーザ名にあたる部分は <token> がセットされます。

store 命令は何らかの問題が Docker Engine で発生した時、STDOUT（標準出力に）エラーを表示できます。

get 命令は STDIN（標準入力）からの文字列をペイロードとして読み込みます。Docker Engine が必要とする証明書を持っているサーバのアドレスをペイロードで渡します。 https://index.docker.io/v1 はペイロードの例です。

get 命令は JSON ペイロードを STDOUT（標準出力）に書き出します。Docker は、このペイロードからユーザ名とパスワードを読み込みます。読み込みます。

```
{
     "Username": "david",
     "Secret": "passw0rd1"
}
```

erase 命令は STDIN（標準入力）からの文字列をペイロードとして読み込みます。Docker Engine が必要とする証明書を持っているサーバのアドレスをペイロードで渡します。 https://index.docker.io/v1 はペイロードの例です。

store 命令は何らかの問題が Docker Engine で発生した時、STDOUT（標準出力に）エラーを表示できます。

ログアウト
logout

使い方: docker logout [サーバ]

Docker レジストリからログアウトする
サーバの指定が無ければデフォルトで "https://index.docker.io/v1/" を使用

--help 使い方の表示

例：

```
$ docker logout localhost:8080
```

pull（プル）

使い方: docker pull [オプション] 名前[:タグ] | [レジストリ・ホスト[:レジストリ・ポート]/]名前[:タグ]

レジストリからイメージやリポジトリを取得

```
-a, --all-tags=false           リポジトリでタグ付けられた全てのイメージをダウンロード
--disable-content-trust=true   イメージの認証をスキップ
--help                         使い方の表示
```

大部分のイメージは、Docker Hub レジストリにあるベース・イメージを元に作られています。

Docker Hub には多くの構築済みのイメージがあります。自分で定義や設定をしなくても、イメージを取得（ダウンロード）して試せます。特定のイメージやイメージの集まり（例：リポジトリ）をダウンロードするには、docker pull を使います。

例

イメージを Docker Hub から取得（pull）

特定のイメージやイメージ群（例：リポジトリ）をダウンロードするには、docker pull を使います。タグを指定しなければ、Docker Engine はデフォルトで :latest タグを使います。次のコマンドは debian:latest イメージを取得します：

```
$ docker pull debian

Using default tag: latest
latest: Pulling from library/debian
fdd5d7827f33: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:e7d38b3517548a1c71e41bffe9c8ae6d6d29546ce46bf62159837aad072c90aa
Status: Downloaded newer image for debian:latest
```

Docker イメージは複数のレイヤ（層）で構成されています。上の例では、イメージには fdd5d7827f33 と a3ed95caeb02 の２つのレイヤがあります。

レイヤはイメージ間で再利用できます。例えば、debian:jessi イメージは debian:latest とレイヤを共有しています。そのため、debian:jessie イメージの取得時は、レイヤをダウンロードせずにメタデータのみ取得します。なぜなら全てのレイヤがローカルにダウンロード済みだからです。

```
$ docker pull debian:jessie

jessie: Pulling from library/debian
fdd5d7827f33: Already exists
a3ed95caeb02: Already exists
Digest: sha256:a9c958be96d7d40df920e7041608f2f017af81800ca5ad23e327bc402626b58e
Status: Downloaded newer image for debian:jessie
```

どのようなイメージがローカルにあるかを確認するには docker images コマンドを使います。

```
$ docker images

REPOSITORY   TAG      IMAGE ID       CREATED      SIZE
debian       jessie   f50f9524513f   5 days ago   125.1 MB
debian       latest   f50f9524513f   5 days ago   125.1 MB
```

Docker はコンテント・アドレッサブル（content-addressable；内容に対して割り当て可能な）イメージ・ストアを使います。そして、イメージ ID とはイメージの設定とレイヤを扱う SHA256 ダイジェストです。先ほどの例では debian:jessie と debian:latest は同じイメージ ID を持っています。イメージ名は違いますが、実際には同じイメージに対してタグ付けしています。どちらも同じイメージのため、レイヤのためのデータを保存するのは 1 度だけであり、余分なディスク容量は不要です。

イメージ、レイヤ、コンテント・アドレッサブル・ストアに関する詳しい情報は、「イメージ、コンテナ、ストレージ・ドライバの理解」のセクションをご覧ください。

ダイジェスト（変わらない識別子）でイメージを取得

ここまではイメージを名前（または「タグ」）で取得しました。イメージを扱うのに名前とタグの指定は便利です。イメージに対して docker pull を実行する時にタグを指定したら、そのイメージの最新バージョンをダウンロードします。例えば docker pull ubuntu:14.04 は Ubuntu 14.04 イメージの最新バージョンを取得します。

イメージを最新バージョンではなく、特定のバージョンに固定したい場合があるでしょう。そのような場合、Docker はダイジェスト（digest 値）を指定してイメージを取得できます。ダイジェストを指定してイメージを取得しようとしたら、指定したバージョンのイメージを確実にダウンロードします。このようするとイメージのバージョンを「固定」し、常に同じイメージの使用を保証します。

イメージのダイジェスト値を知るには、まずイメージを取得します。Docker Hub から最新の ubuntu:14.04 イメージをダウンロードしましょう。

```
$ docker pull ubuntu:14.04

14.04: Pulling from library/ubuntu
5a132a7e7af1: Pull complete
fd2731e4c50c: Pull complete
28a2f68d1120: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2
Status: Downloaded newer image for ubuntu:14.04
```

Docker はダウンロードが完了したら、イメージのダイジェスト値を表示します。先ほどの例では、イメージのダイジェスト値とは、こちらです。

```
sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2
```

Docker はイメージを送信（push）する時のダイジェスト値を表示します。イメージを送信時のバージョンを固定したい場合には便利になるでしょう。

イメージの取得時にダイジェスト値を使うには、タグとして扱います。例えば、イメージをダイジェスト値で取得するには、次のコマンドを実行します。

```
$ docker pull ubuntu@sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2

sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2: Pulling from library/ubuntu
5a132a7e7af1: Already exists
fd2731e4c50c: Already exists
28a2f68d1120: Already exists
a3ed95caeb02: Already exists
Digest: sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2
Status: Downloaded newer image for
ubuntu@sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2
```

Digest は Dockerfile の FROM でも指定可能です。以下は例です。

```
FROM ubuntu@sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2
MAINTAINER some maintainer <maintainer@example.com>
```

この機能はイメージに対するバージョンを都度「固定」します。そのため Docker はイメージのバージョンを更新しないため、セキュリティの更新もしません。更新版のイメージを取得したい場合は、適時ダイジェスト値を変更する必要があります。

別のレジストリから取得

docker pull のイメージは Docker Hub から取得するのがデフォルトです。取得するレジストリの場所は、手動で指定可能です。例えば、ローカルにレジストリをセットアップしておけば、そちらを指定してイメージを取得できます。レジストリのパスは URL と似ていますが、プロトコル指示子（https://）がありません。

以下のコマンドは、ポート 5000 を開いているローカルのレジストリ（myregistry.local:5000）から、testing/test-image イメージを取得するコマンドです。

```
$ docker pull myregistry.local:5000/testing/test-image
```

レジストリの認証情報は docker login で管理します。

Docker はレジストリとの通信に https プロトコルを使います。ただし、レジストリが安全ではない接続 (insecure connection）を許可している場合は除外します。詳細は「安全ではないレジストリ」のセクションをご覧ください。

リポジトリから複数のイメージを取得

デフォルトでは、docker pull はレジストリから単一のイメージを取得します。リポジトリには複数のイメージがあります。リポジトリから全てのイメージを取得するには docker pull で -a（あるいは --all-tags ）オプションを使います。

次のコマンドは fedora リポジトリから全てのイメージを取得します。

```
$ docker pull --all-tags fedora

Pulling repository fedora
ad57ef8d78d7: Download complete
105182bb5e8b: Download complete
511136ea3c5a: Download complete
73bd853d2ea5: Download complete
....

Status: Downloaded newer image for fedora
```

取得が終わったら、取得した全てのイメージを確認するために docker images コマンドを使います。次の例はローカルに現在ある全ての fedora イメージを表示しています。

```
$ docker images fedora

REPOSITORY   TAG         IMAGE ID       CREATED      SIZE
fedora       rawhide     ad57ef8d78d7   5 days ago   359.3 MB
fedora       20          105182bb5e8b   5 days ago   372.7 MB
fedora       heisenbug   105182bb5e8b   5 days ago   372.7 MB
fedora       latest      105182bb5e8b   5 days ago   372.7 MB
```

取得を中止

docker pull プロセスを停止するには、ターミナルで実行中に CTRL-c を押すと、pull 処理を中断します。

```
$ docker pull fedora

Using default tag: latest
latest: Pulling from library/fedora
a3ed95caeb02: Pulling fs layer
236608c7b546: Pulling fs layer
^C
```

技術的に Engine を停止する処理とは、Docker Engine デーモンと起点となった Docker Engine クライアント間における取得（pull）に対してです。何らかの理由によって Engine デーモンとの通信を切断した場合も、同様に取得処理が中断します。

push（プッシュ）

使い方: docker push [オプション] 名前[:タグ]

イメージやリポジトリをレジストリに送信（push）

```
--disable-content-trust=true   イメージの署名をスキップ
--help                         使い方の表示
```

docker push を使うと、イメージを Docker Hub レジストリや、自分で作成したレジストリで共有できるようになります。

docker push プロセスを停止するには、ターミナルで実行中に CTRL-c を押すと、push 処理を中断します。

レジストリの認証情報は docker login で管理します。

search（サーチ）

使い方: docker search [オプション] 単語

Docker Hub のイメージを検索

--automated	自動構築（automated build）のみ表示
--help	使い方の表示
--no-trunc	トランケート（truncate）を出力しない
-s, --stars=0	最低限 x 個のスターがあるイメージのみ表示

Docker Hub のイメージを検索します。

共有イメージをコマンドラインで調べる詳細は、「Docker Hub で公開イメージを探す」セクションをご覧ください。

検索結果は 25 件までしか表示しません。

A2.7 ネットワークと接続用コマンド

network connect（コネクト）

使い方: docker network connect [オプション] ネットワーク コンテナ

コンテナをネットワークに接続

--alias=[]	コンテナ用のネットワーク範囲エイリアスを追加
--help	使い方の表示
--ip	IPv4 アドレス
--ip6	IPv6 アドレス
--link=[]	他のコンテナに対するリンクを追加

コンテナをネットワークに接続（connect）します。コンテナの接続はコンテナ名かコンテナ ID を使います。接続後は、同一ネットワーク上にある他のコンテナと通信可能になります。

```
$ docker network connect multi-host-network container1
```

あるいは、docker run --net=<ネットワーク名> オプションを使えば、コンテナ起動時に直ちにネットワークに接続します。

```
$ docker run -itd --net=multi-host-network busybox
```

コンテナのインターフェースに任意の IP アドレスを割り当て可能です。

```
$ docker network connect --ip 10.10.36.122 multi-host-network container2
```

--link オプションを使うことで、他のコンテナを任意のエイリアス名でリンクできます。

```
$ docker network connect --link container1:c1 multi-host-network container2
```

--alias オプションを使うことで、ネットワークを接続したコンテナ間での名前解決に使う別名を指定できます。

```
$ docker network connect --alias db --alias mysql multi-host-network container2
```

コンテナを中断（pause）・再起動・停止しても、ネットワークに接続したままです。中断したコンテナはネットワークに接続し続けており、network inspect で確認できます。コンテナを停止（stop）すると、再起動するまではネットワーク上に表示されません。

停止しているコンテナを再起動する時に IP アドレスを指定できます。もしも IP アドレスが使えなければ、コンテナは起動に失敗します。IP アドレスを確実に割り当てるためには、ネットワーク作成時に --ip-range（IP アドレスの範囲）を指定しておき、その範囲内外にある静的な IP アドレスを割り当てる方法があります。そうしておけば、コンテナが対象のネットワークに所属していない間でも、他のコンテナに IP アドレスを使われる心配はありません。

```
$ docker network create --subnet 172.20.0.0/16 --ip-range 172.20.240.0/20 multi-host-network

$ docker network connect --ip 172.20.128.2 multi-host-network container2
```

コンテナがどこに接続しているかを確認するには、docker network inspect コマンドを使います。docker network

`disconnect` はコンテナをネットワークから切り離します。

ネットワークに接続したら、コンテナは他のコンテナの IP アドレスや名前を使って通信できるようになります。overlay ネットワークやカスタム・プラグインは複数のホスト間の接続性（multi-host connectivity）をサポートしています。コンテナは同じマルチホスト・ネットワーク上で接続できるだけではありません。異なったエンジンによって起動されていたとしても、同様に通信できます。

コンテナは複数のネットワークにも接続できます。ネットワークは同じ種類でなくても構いません。例えば、コンテナ・ブリッジとオーバレイ・ネットワークの両方に接続できます。

network create（クリエイト）

使い方: docker network create [オプション] ネットワーク名

ユーザが名付ける新しいネットワークを作成

--aux-address=map[]	ネットワーク・ドライバが使う ipv4 または ipv6 追加アドレス
-d --driver=DRIVER	ネットワーク・ブリッジまたはオーバレイを管理するドライバ。デフォルトは bridge
--gateway=[]	マスタ・サブネット用の ipv4 または ipv6 ゲートウェイ
--help	使い方の表示
--internal	ネットワークの外部へのアクセスを制限
--ip-range=[]	サブ・レンジからコンテナの IP アドレスを割り当て
--ipam-driver=default	IP アドレス管理用ドライバ
--ipam-opt=map[]	カスタム IPAM ドライバ固有のオプションを指定
-o --opt=map[]	カスタムドライバ固有のオプションを指定
--subnet=[]	ネットワーク・セグメントを表すサブネットを CIDR 形式で指定

新しいネットワークを作成します。DRIVER には bridge か overlay を指定できます。どちらも内蔵のネットワーク・ドライバです。サードパーティー製のドライバをインストールするか、カスタム・ネットワーク・ドライバを組み込むのであれば、同様に DRIVER で指定できます。 --driver オプションを指定しなければ、コマンドは自動的に bridge ネットワークを作成します。Docker エンジンをインストールしたら、bridge ネットワークを自動的に構築します。このネットワークは従来の docker0 ブリッジに相当します。新しいコンテナを docker run で起動したら、自動的にこのブリッジ・ネットワークに接続します。このデフォルト・ブリッジ・ネットワークは削除できませんが、新しいブリッジを network create コマンドで作成できます。

```
$ docker network create -d bridge my-bridge-network
```

ブリッジ・ネットワークは、単一の Docker エンジン上でネットワークを分離（isolate）します。ネットワークを作成する時、overlay（オーバレイ）ネットワークであれば、Docker エンジンが動作する複数のホスト上に渡ることも可能です。bridge ネットワークとは異なり、オーバレイ・ネットワークを作成するには、いくつかの条件が必要です。

- キーバリュー・ストアにアクセスできること。エンジンがサポートしているキーバリュー・ストアは Consul、Etcd、ZooKeeper（分散ストア）です。
- クラスタの各ホストが、キーバリュー・ストアと接続できること。
- 各ホスト上の Docker エンジンの daemon が、クラスタとしての適切な設定をすること。

docker daemon が overlay ネットワークをサポートするために必要なオプションは、次の通りです。

- --cluster-store
- --cluster-store-opt
- --cluster-advertise

各オプションや設定方法の詳細については、「マルチホスト・ネットワークを始めましょう」のセクションをご覧ください。

もう 1 つの良いアイディアとしては、これらの設定を行わなくても、Docker Swarm でネットワークも管理できるクラスタの管理もできます。Swarm は知的なディスカバリとサーバ管理を提供しますので、実装にあたっての手助けとなるでしょう。

overlay ネットワークに必要な準備が整った後は、クラスタ上にあるホストのどれかで次のようなネットワーク作成コマンドを実行します。

```
$ docker network create -d overlay my-multihost-network
```

ネットワーク名はユニークにする必要があります。Docker デーモンは名前の衝突を避けようとしますが、保証されません。ユーザ・リポジトリも名前の衝突を避けようとします。

コンテナに接続

コンテナの起動時に --net フラグを指定したら、ネットワークに接続します。busybox コンテナが mynet ネットワークに接続するには、次のようにします。

```
$ docker run -itd --net=mynet busybox
```

既に実行中のコンテナに対してネットワークを接続したい場合は、docker network connect サブコマンドを使います。

同じネットワークに複数のコンテナが接続できます。接続したら、コンテナは他のコンテナの IP アドレスか名前で通信できるようになります。overlay ネットワークやカスタム・プラグインは、複数のホスト間の接続サポートしていますので、コンテナは同じマルチホスト・ネットワークに接続できるだけでなく、異なった Docker エンジンから起動された環境でも、同様に通信できます。

コンテナをネットワークから切断するには、docker network disconnect コマンドを使います。

高度なオプションの設定

ネットワークの作成時、デフォルトではエンジンはネットワークのサブネットワークが重複しないようにします。サブネットワークは既存のネットワークの下位にはありません。純粋に IP アドレスを割り当てるためです。このデフォルトを上書きするには、 --subnet オプションを使ってサブネットワークの値を直接指定します。

```
docker network create -d --subnet=192.168.0.0/16
```

更に、他にも --gateway --ip-range --aux-address オプションが利用可能です。

```
network create --driver=bridge --subnet=172.28.0.0/16 --ip-range=172.28.5.0/24 \
  --gateway=172.28.5.254 br0
```

--gateway フラグを省略したら、エンジンは対象ネットワークの範囲内から 1 つ選びます。overlay ネットワークとネットワーク・ドライバ・プラグインの場合は、複数のサブネットワークの作成をサポートしています。

```
docker network create -d overlay
  --subnet=192.168.0.0/16 --subnet=192.170.0.0/16
  --gateway=192.168.0.100 --gateway=192.170.0.100
  --ip-range=192.168.1.0/24
  --aux-address a=192.168.1.5 --aux-address b=192.168.1.6
  --aux-address a=192.170.1.5 --aux-address b=192.170.1.6
  my-multihost-network
```

サブ・ネットワークが重複しないように気を付けてください。重複したらネットワークの作成に失敗し、エンジンはエラーを表示します。

ブリッジ・ドライバのオプション

カスタム・ネットワークの作成時、デフォルトのネットワーク・ドライバ（例： bridge ）では追加のオプションを指定できます。以下のオプション指定は、 docker デーモンで docker0 ブリッジ用のフラグを指定するのと同等です。

オプション	同等	説明
com.docker.network.bridge.name	−	Linux ブリッジを作成する時に使うブリッジ名
com.docker.network.bridge.enable_ip_masquerade	--ip-masq	IP マスカレードの有効化
com.docker.network.bridge.enable_icc	--icc	内部におけるコンテナの接続性を、有効化または無効化
com.docker.network.bridge.host_binding_ipv4	--ip	コンテナのポートをバインドする時の、デフォルト IP アドレスを指定
com.docker.network.bridge.mtu	--mtu	コンテナのネットワーク MTU を指定

以下の引数は docker network create 実行時、あらゆるネットワーク・ドライバで指定できます。ほとんどが docker daemon で指定する項目と同等です。

オプション	同等	説明
--geteway	−	マスタ・サブネットに対する IPv4 または IPv6 ゲートウェイ
--ip-range	--fixed-cidr	範囲内で割り当てる IP アドレス
--internal	−	外部ネットワークに対する接続を制限
--ipv6	--ipv6	IPv6 ネットワーク機能を有効化
--subnet	--bip	ネットワーク用のサブネット

例えば、ポート公開用に使う IP アドレスを割り当てるには、 -o か --opt オプションを使います。

```
docker network create -o "com.docker.network.bridge.host_binding_ipv4"="172.19.0.1" simple-network
```

内部ネットワーク(internal)モード

コンテナを overlay ネットワークに接続する時、デフォルトでは外部への接続性を提供するためブリッジ・ネットワークにも接続します。外部された隔離された overlay ネットワークを作成したい場合は、 --internal オプションを使います。

network disconnect（ディスコネクト）

使い方: docker network disconnect [オプション] ネットワーク コンテナ

ネットワークからコンテナを切断

```
  -f, --force        ネットワークからコンテナを強制切断
  --help             使い方の表示
```

コンテナをネットワークから切り離します（disconnect)。ネットワークから切り離すためには、コンテナを実行している必要があります。

```
$ docker network disconnect multi-host-network container1
```

network inspect（インスペクト）

使い方: docker network inspect [オプション] ネットワーク [ネットワーク..]

ネットワークの詳細情報を表示

```
  -f, --format=       go テンプレートで指定したフォーマットで表示
  --help              使い方の表示
```

1つまたは複数のネットワークの情報を返します。デフォルトでは、このコマンドは結果を JSON オブジェクト形式で返します。例えば、2つのコンテナをデフォルトの bridge ネットワークに接続します。

```
$ sudo docker run -itd --name=container1 busybox
f2870c98fd504370fb86e59f32cd0753b1ac9b69b7d80566ffc7192a82b3ed27

$ sudo docker run -itd --name=container2 busybox
bda12f8922785d1f160be70736f26c1e331ab8aaf8ed8d56728508f2e2fd4727
```

network inspect コマンドで、その結果からコンテナの情報を確認します。ネットワーク・バックエンドが Overlay のようなマルチホスト・ネットワーク・ドライバの場合は、コマンドはクラスタ上の他のホストに対するコンテナのエンドポイントも表示します。これらのエンドポイントは“ ep-{エンドポイント-id}”として表示されます。あるいは、結果表示の時に別のフォーマットも指定できます。フォーマットの詳細は Go 言語の text/template パッケージで指定します。

```
$ sudo docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "b2b1a2cba717161d984383fd68218cf70bbbd17d328496885f7c921333228b0f",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.17.42.1/16",
                    "Gateway": "172.17.42.1"
                }
            ]
        },
        "Internal": false,
        "Containers": {
            "bda12f8922785d1f160be70736f26c1e331ab8aaf8ed8d56728508f2e2fd4727": {
                "Name": "container2",
                "EndpointID": "0aebb8fcd2b282abe1365979536f21ee4ceaf3ed56177c628eae9f706e00e019",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            },
            "f2870c98fd504370fb86e59f32cd0753b1ac9b69b7d80566ffc7192a82b3ed27": {
                "Name": "container1",
                "EndpointID": "a00676d9c91a96bbe5bcfb34f705387a33d7cc365bac1a29e4e9728df92d10ad",
                "MacAddress": "02:42:ac:11:00:01",
                "IPv4Address": "172.17.0.1/16",
```

```
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        }
    }
]
```

ユーザ定義ネットワークに関する詳細情報を返します。

```
$ docker network create simple-network
69568e6336d8c96bbf57869030919f7c69524f71183b44d80948bd3927c87f6a
$ docker network inspect simple-network
[
    {
        "Name": "simple-network",
        "Id": "69568e6336d8c96bbf57869030919f7c69524f71183b44d80948bd3927c87f6a",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.22.0.0/16",
                    "Gateway": "172.22.0.1/16"
                }
            ]
        },
        "Containers": {},
        "Options": {}
    }
]
```

network ls

使い方: docker network ls [オプション]

ユーザが作成した全てのネットワークを一覧
-f, --filter=[] 指定した状況に応じて出力をフィルタ
--help 使い方の表示
--no-trunc 出力を省略（truncate）しない
-q, --quiet 整数値の ID のみ表示

Docker エンジンの daemon が把握している全てのネットワーク一覧を表示します。ネットワークには、複数のホストによるクラスタ上にまたがるネットワークも含まれます。

```
$ sudo docker network ls
NETWORK ID          NAME                DRIVER
7fca4eb8c647        bridge              bridge
9f904ee27bf5        none                null
cf03ee007fb4        host                host
78b03ee04fc4        multi-host          overlay
```

--no-trunk オプションを使えば、完全なネットワーク ID を表示します。

```
docker network ls --no-trunc
NETWORK ID                                                         NAME                DRIVER
18a2866682b85619a026c81b98a5e375bd33e1b0936a26cc497c283d27bae9b3   none                null
c288470c46f6c8949c5f7e5099b5b7947b07eabe8d9a27d79a9cbf111adcbf47   host                host
7b369448dccbf865d397c8d2be0cda7cf7edc6b0945f77d2529912ae917a0185   bridge              bridge
95e74588f40db048e86320c6526440c504650a1ff3e9f7d60a497c4d2163e5bd   foo                 bridge
63d1ff1f77b07ca51070a8c227e962238358bd310bde1529cf62e6c307ade161   dev                 bridge
```

フィルタリング

フィルタリング・フラグ（ -f または --filter）の書式は key=value のペアです。フィルタを何回もしたい場合は、複数のフラグを使います（例： -filter "foo=bar" --filter "bif=baz" ）。複数のフィルタを指定したら、OR（同一条件）フィルタとして連結されます。例えば、 -f type=custom -f type=builtin は custom と builtin ネットワークの両方を返します。

現時点でサポートしているフィルタは、次の通りです。

- ID（ネットワーク ID）
- ラベル（ label=<キー> または label=<キー>=<値> ）
- 名前（ネットワーク名）
- タイプ（custom|builtin）

id

id フィルタはネットワーク ID の一部もしくは全体と一致します。

以下のフィルタは、コンテナ ID が 63d1ff1f77b0...文字列に一致する全てのネットワークを表示します。

```
$ docker network ls --filter id=63d1ff1f77b07ca51070a8c227e962238358bd310bde1529cf62e6c307ade161
NETWORK ID          NAME                DRIVER
63d1ff1f77b0        dev                 bridge
```

次のように ID の部分一致でもフィルタできます。

```
$ docker network ls --filter id=95e74588f40d
NETWORK ID          NAME                DRIVER
95e74588f40d        foo                 bridge

$ docker network ls --filter id=95e
NETWORK ID          NAME                DRIVER
95e74588f40d        foo                 bridge
```

ラベル

label フィルタは label だけ、あるいは label と値に一致する条件でフィルタします。
以下のフィルタはラベルの値が usage に一致するネットワークを表示します。

```
$ docker network ls -f "label=usage"
NETWORK ID          NAME                DRIVER
db9db329f835        test1               bridge
f6e212da9dfd        test2               bridge
```

以下のフィルタは usage ラベルの値が prod の値に一致するコンテナを表示します。

```
$ docker network ls -f "label=usage=prod"
NETWORK ID          NAME                DRIVER
f6e212da9dfd        test2               bridge
```

名前

name フィルタはネットワーク名の一部もしくは全体に一致します。
以下のフィルタは foobar 文字列を含む全てのネットワーク名でフィルタします。

```
$ docker network ls --filter name=foobar
NETWORK ID          NAME                DRIVER
06e7eef0a170        foobar              bridge
```

次のように、部分一致でもフィルタできます。

```
$ docker network ls --filter name=foo
NETWORK ID          NAME                DRIVER
95e74588f40d        foo                 bridge
06e7eef0a170        foobar              bridge
```

タイプ

type フィルタは２つの値をサポートしています。builtin は定義済みネットワーク（bridge、none、host）を表示します。custom はユーザ定義ネットワークを表示します。
以下のフィルタはユーザ定義ネットワークを全て表示します。

```
$ docker network ls --filter type=custom
NETWORK ID          NAME                DRIVER
95e74588f40d        foo                 bridge
63d1ff1f77b0        dev                 bridge
```

このフラグを指定したら、バッチ処理でクリーンアップできます。例えば、全てのユーザ定義ネットワークを削除するには、次のようにします。

```
$ docker network rm `docker network ls --filter type=custom -q`
```

コンテナがアタッチされているネットワークを削除しようとしたら、警告が表示されます。

network rm

使い方: docker network rm [オプション] 名前|ID

ネットワークを削除

--help 使い方の表示

ネットワーク名か ID を指定して削除します。ネットワークを削除するには、接続中の全てのコンテナとの接続を切断しなくてはいけません。

```
$ docker network rm my-network
```

複数のネットワークを削除する場合は、docker network rm コマンドで複数のネットワーク名や ID を指定できます。以下の例はネットワーク ID 3695c422697f とネットワーク名 my-network を削除します。

```
$ docker network rm 3695c422697f my-network
```

複数のネットワークを指定すると、コマンドは削除したネットワークの情報を返します。もしも、あるネットワークの情報削除に失敗すると、コマンドはリスト上の次にあるネットワークを削除しようとします。コマンドは削除に成功したか失敗したかを表示します。

A2.8 共有データ・ボリューム用コマンド

volume create（クリエイト）

使い方: docker volume create [オプション]

ボリュームを作成

```
  -d, --driver=local    ボリューム・ドライバ名を指定
  --help                使い方の表示
  --label=[]            ボリューム用のメタデータを指定
  --name=               ボリューム名を指定
  -o, --opt=map[]       ドライバ固有のオプションを指定
```

コンテナが利用し、データを保管するための新しいボリュームを作成します。名前を指定しなければ、Dockerはランダムな名称を生成します。ボリュームを作成し、コンテナが使えるようにするには、次の例のように実行します。

```
$ docker volume create --name hello
hello

$ docker run -d -v hello:/world busybox ls /world
```

これはコンテナ内の /world ディレクトリにマウントを作成します。Docker はコンテナ内のマウントポイントに、相対パスの指定をサポートしません。

複数のコンテナが同時に同じボリュームを利用できます。２つのコンテナが共有データにアクセスする場合に便利です。例えば、一方のコンテナがデータを書き込み、もう一方がデータを読み込めます。

ボリューム名はドライバ間でユニークである必要があります。つまり２つの異なったドライバで、同じボリューム名を使うことはできません。実行しようとしても、docker はエラーを返します。

```
A volume named  %s  already exists with the %s driver. Choose a different volume name.
```

現在のドライバが既に使用しているボリューム名を指定する場合は、Docker は既存のボリュームを再利用するとみなし、エラーを返しません。

ドライバ固有のオプション

ボリューム・ドライバによっては、ボリューム作成のカスタマイズのためにオプションが使えます。 -o か --opt フラグでドライバにオプションを渡します。

```
$ docker volume create --driver fake --opt tardis=blue --opt timey=wimey
```

これらのオプションは各ボリューム・ドライバに直接渡されます。オプションはボリューム・ドライバごとに別々の動作をします（あるいは何もしないかもしれません）。

内部の local ボリューム・ドライバは、現時点ではオプションを受け付けません。

volume inspect（インスペクト）

使い方: docker volume inspect [オプション] ボリューム [ボリューム...]

ボリュームの低レベル情報を返す

```
  -f, --format=       指定した go テンプレートの形式で出力
  --help              使い方の表示
```

ボリュームに関する情報を返します。デフォルトでは、コマンドは JSON 配列の形式です。実行テンプレートを個々に指定し、別のフォーマットを指定できます。Go 言語の text/template パッケージに、フォーマットの詳細に関する全てが記述されています。

出力例：

```
$ docker volume create
85bffb0677236974f93955d8ecc4df55ef5070117b0e53333cc1b443777be24d
$ docker volume inspect 85bffb0677236974f93955d8ecc4df55ef5070117b0e53333cc1b443777be24d
[
  {
      "Name": "85bffb0677236974f93955d8ecc4df55ef5070117b0e53333cc1b443777be24d",
      "Driver": "local",
      "Mountpoint":
"/var/lib/docker/volumes/85bffb0677236974f93955d8ecc4df55ef5070117b0e53333cc1b443777be24d/_data",
      "Status": null
  }
]

$ docker volume inspect --format '{{ .Mountpoint }}'
85bffb0677236974f93955d8ecc4df55ef5070117b0e53333cc1b443777be24d
/var/lib/docker/volumes/85bffb0677236974f93955d8ecc4df55ef5070117b0e53333cc1b443777be24d/_data
```

volume ls

使い方: docker volume ls [オプション]

ボリュームの一覧

```
-f, --filter=[]      次の状況に応じて出力フィルタ:
                      - dangling=<boolean> ボリュームが参照されているかどうか
                      - driver=<string> ボリュームのドライバ名
                      - name=<string> ボリューム名
--help               使い方の表示
-q, --quiet          ボリューム名のみ表示
```

Docker が把握している全てのボリュームを表示します。 `-f` か `--filter` フラグを使ってフィルタできます。利用可能なフィルタ・オプションに関する情報は「フィルタリング」のセクションをご覧ください。

出力例：

```
$ docker volume create --name rose
rose
$docker volume create --name tyler
tyler
$ docker volume ls
DRIVER              VOLUME NAME
local               rose
local               tyler
```

フィルタリング

フィルタリング・フラグ（ `-f` か `--filter`）は「key=value」の形式です。フィルタが複数ある場合は、複数回指定します（例： `--filter "foo=bar" --filter "bif=baz"` ）。

現時点でサポートしているフィルタ：

- ダングリング（ブール値 - true か false か 0 か 1 ）
- ドライバ（ボリュームのドライバ名）
- 名前（ボリューム名）

ダングリング

`dangling` フィルタはコンテナから参照されていない（dangling ＝宙ぶらりんな）ボリュームに一致します。

```
$ docker run -d  -v tyler:/tmpwork  busybox
f86a7dd02898067079c99ceacd810149060a70528eff3754d0b0f1a93bd0af18
$ docker volume ls -f dangling=true
DRIVER              VOLUME NAME
local               rosemary
```

ドライバ

`driver` フィルタはボリュームのドライバ名の全てまたは一部に一致します。

以下のフィルタはドライバ名に `local` 文字列を含む全てのボリュームを表示します。

```
$ docker volume ls -f driver=local
DRIVER              VOLUME NAME
local               rosemary
local               tyler
```

名前

name フィルタはボリューム名の全てまたは一部に一致します。

以下のフィルタはボリューム名に rose 文字列を含む全てのボリュームを表示します。

```
$ docker volume ls -f name=rose
DRIVER              VOLUME NAME
local               rosemary
```

volume rm

使い方: docker volume rm [オプション] ボリューム [ボリューム...]

ボリュームを削除

--help=false 使い方の表示

1 つまたは複数のボリュームを削除できます。コンテナが使用中のボリュームは削除できません。

```
$ docker volume rm hello
hello
```

Appendix.3

Dockerfile リファレンス

A3.1 Dockerfile

Docker は Dockerfile から命令を読み込み、自動的にイメージを構築できます。Dockerfile はテキスト形式のドキュメントであり、コマンドライン上でイメージを作り上げる命令を全て記述します。ユーザは docker build を使い、複数のコマンド行の命令を順次実行し、イメージを自動構築します。

このセクションでは Dockerfile 内で利用可能な命令を説明します。セクションを読み終えたら、より便利に使うために「Dockerfile のベスト・プラクティス」のセクションもご覧ください。

使い方

docker build コマンドは Dockerfile とコンテクスト(context；イメージに含まれる「内容」の意味）に従ってイメージを構築します。構築用コンテクストとは、ファイルを示す PATH や URL の場所です。PATH はローカルのファイルシステム上のディレクトリです。URL は Git リポジトリの場所です。

コンテクストの処理は再帰的です。そのため、PATH にはサブディレクトリを含みます。また URL であればリポジトリと、そのサブモジュールも含みます。単に build コマンドを実行したら、現在のディレクトリをコンテクストとして使います。

```
$ docker build .
Sending build context to Docker daemon  6.51 MB
...
```

構築を処理するのは Docker デーモンであり、CLI ではありません。まずはじめの構築プロセスは、対象のコンテクスト（再帰的）をデーモンに送信することです。多くの場合、空のディレクトリをコンテクストとして使いますので、Dockerfile をそのディレクトリ設置できます。Dockerfile の構築に必要なファイルのみを（ディレクトリに）追加します。

【警告】PATH として自分のルート・ディレクトリ / を使わないでください。これは、自分のハードディスクに含まれる内容を、Docker デーモンに転送しようとするためです。

Dockerfile に記述した COPY 命令などで使うファイル指定を参照し、コンテクスト（内容物の意味）を構築します。構築パフォーマンスを向上するためには、 .dockerignore ファイルにファイルやディレクトリを追加し、コンテクスト・ディレクトリから除外できます。より詳しい情報は、「.dockerignore ファイルの作成」セクションををご覧ください。

伝統的に Dockerfile は、Dockerfile とコンテクストがあるルートの場所を示します。docker build 時に -f フラグを使えば、システム上のどこに Dockerfile があるか指定できます。

```
$ docker build -f /path/to/a/Dockerfile .
```

新しいイメージの構築に成功する時は、新しいイメージにリポジトリとタグを指定できます。

```
$ docker build -t shykes/myapp .
```

Docker デーモンは Dockerfile の命令を 1 行ずつ実行し、必要があれば命令ごとにイメージをコミットし、最終的に新しいイメージ ID を出力します。Docker デーモンは送信したコンテクストを自動的に削除します。

各命令は独立して実行されるのでご注意ください。新しいイメージの作成時、RUN cd /tmp を実行したとしても、次の命令には何ら影響を与えません。

Docker は可能であればいつでも中間イメージ（キャッシュ）を再利用します。これは docker build 処理を速くするためです。コンソール出力に Using cache（キャッシュを利用中）の文字列が表示されます。より詳しい情報は Dockerfile ベスト・プラクティス・ガイドの「構築キャッシュ」のセクションをご覧ください。

```
$ docker build -t svendowideit/ambassador .
Sending build context to Docker daemon 15.36 kB
Step 0 : FROM alpine:3.2
 ---> 31f630c65071
Step 1 : MAINTAINER SvenDowideit@home.org.au
 ---> Using cache
 ---> 2a1c91448f5f
Step 2 : RUN apk update &&       apk add socat &&         rm -r /var/cache/
 ---> Using cache
 ---> 21ed6e7fbb73
Step 3 : CMD env | grep _TCP= | sed 's/.*_PORT_\([0-9]*\)_TCP=tcp:\/\/\(.*\):\(.*\)/socat -t 100000000
TCP4-LISTEN:\1,fork,reuseaddr TCP4:\2:\3 \&/' && echo wait) | sh
 ---> Using cache
 ---> 7ea8aef582cc
Successfully built 7ea8aef582cc
```

構築が終わったら、レジストリにリポジトリを送信する準備が整います。

書式

ここでは Dockerfile の書式を説明します。

```
# コメント
命令 引数
```

命令（instruction）は大文字と小文字を区別しません。しかし**引数（arguments）**を簡単に見分けられるよう、大文字にするのが便利です。

Docker は Dockerfile の命令を順番に実行します。イメージ構築にあたりベース・イメージを指定するため、**1 行めの命令は「FROM」であるべき**です。

Docker は # で始まる行をコメントとみなします。 # マークは行における移行の文字をコメントとみなします。コメントは次のような書き方ができます。

```
# コメント
RUN echo '何か良いものを # で実行しています'
```

ここでは、Dockerfile でイメージ構築時に利用可能な命令セットを紹介します。

環境変数の置き換え

Dockerfile は環境変数（env 命令で宣言）も解釈できます。命令文字（ステートメント・リテラル）中では、変数の様な構文でエスケープ・シーケンスも扱えます。

Dockerfile の中では、環境変数を$variable_name または${variable_name}の形式で記述します。これらは同等に扱われます。固定用の構文として典型的に使われるのは、空白スペースを変数名に入れず${foo}_bar のような変数名で割り当てることです。

${変数の_名前} 構文は、次のような bash の変更をサポートしています。

- **${変数:-文字}**は、**変数**を設定したら、その値を使うことを意味します。もし**変数**がセットされなければ、**文字**が設定されます。
- **${変数:+文字}**は、**変数**を設定したら、**文字**を使います。**変数**がセットされなければ、空白のままにします。

いずれの場合でも、**文字**とは何らかの文字列であり、追加の環境変数を含みます。

エスケープするには \$foo や \${foo} のように、変数名の前に \ を付けます。例えば、 $foo と ${foo} リテラルは別々のものです。

例（変数展開したものは、 # のあとに表示）：

```
FROM busybox
ENV foo /bar
WORKDIR ${foo}   # WORKDIR /bar
ADD . $foo       # ADD . /bar
COPY \$foo /quux # COPY $foo /quux
```

以下の命令で Dockerfile における環境変数の利用がサポートされています。

- ADD
- COPY
- ENV
- EXPOSE
- LABEL
- USER
- WORKDIR
- VOLUME
- STOPSIGNAL

同様に、

- ONBLIUD （上記の命令と組み合わせて使う場合にサポートされます）

1.4 より前のバージョンでは、環境変数における ONBUILD 命令と上記の命令の組み合わせはサポートしていません。

環境変数を使う代わりに、各変数をコマンド上で利用できます。次の例を見ましょう。

```
ENV abc=hello
ENV abc=bye def=$abc
ENV ghi=$abc
```

この結果は、def の値が hello であり、bye ではありません。しかしながら ghi の値は bye になります。これは abc を bye に設定するのと同じコマンド行ではないためです。

.dockerignore ファイル

docker CLI がコンテクストを docker デーモンに送る前に、コンテクストのルートディレクトリ内の .dockerignore ファイルを探します。もしファイルが存在していれば、CLI はコンテクストからパターンに一致するファイルとディレクトリを除外します。これは不必要に大きくならないようにします。また、取り扱いに注意が必要なファイルやディレクトリをデーモンに送らないようにします。ですが、ADD や COPY でイメージに追加されるかもしれません。

CLI は .dockerignore ファイルを行ごとに隔てて解釈します。行の一致パターンは Unix シェル上のものに似ています。パターンがコンテクストの root に一致すると考えられる場合は、root ディレクトリとして動作します。例えば、パターン /foo/bar と foo/bar がある場合、いずれも PATH における foo サブディレクトリの bar ファイルを削除します。あるいは URL の場所にある git のルートでもです。どちらでも除外されます。

これは .dockerignore ファイルの例です：

```
*/temp*
*/*/temp*
temp?
```

このファイルは構築時に以下の動作をします。

- */temp* … ルート以下のあらゆるサブディレクトリを含め、temp で始まる名称のファイルとディレクトリを除外します。例えば、テキストファイル /somedir/temporary.txt は除外しますし、ディレクトリ /somedir/temp も除外します。
- */*/temp* … ルートから 2 レベル以下の temp で始まる名称のファイルとディレクトリを除外します。例えば /somedir/subdir/temporary.txt を除外します。
- temp? … ルートディレクトリにあるファイル名が temp と 1 文字一致するファイルとディレクトリを除外します。例えば、 /tempa と /tempb を除外します。

一致には Go 言語の filepath.Match[*1] ルールを使います。処理前のステップでは、空白スペースと . と .. 要素を Go 言語の filepath.Clean[*2] を用いて除外します。

行を !（エクスクラメーション・マーク）で始めたら、除外ルールとして使えます。以下の例は .dockerignore ファイルでこの仕組みを使ったものです。

```
*.md
!README.md
```

README.md を除く全てのマークダウンファイルが、コンテントから除外されます。

! 除外ルールが影響を与えるのは、 .dockerignore ファイルに書いた場所以降に一致するパターンが現れた時、含めるか除外するかを決めます。次の例で考えて見ましょう。

```
*.md
!README*.md
README-secret.md
```

*1 http://golang.org/pkg/path/filepath#Match

*2 http://golang.org/pkg/path/filepath/#Clean

README を含むファイル以外は、README-secret.md も含め、残り全てのマークダウンファイルが除外対象です。
その次の例を考えましょう。

```
*.md
README-secret.md
!README*.md
```

README を含む全てのファイル除外します。真ん中の行 README-secret.md は最終行の !README*.md に一致するため、何の影響もありません。

.dockerignore ファイルは Dockerfile と .dockerignore ファイルの除外にも使えます。それでも、これらのファイルはジョブを処理するためデーモンに送信されます。しかし ADD と COPY コマンドは、これらをイメージ内にコピーしません。

最後に、特定のファイルのみコンテクストに含め、他を除外したい場合があるでしょう。実行するには、始めに* パターンに指定し、以下 1 つまたは複数の ！例外パターンを記述します。

歴史的な理由により、 . パターンは無視されます。

A3.2 Dockerfile 命令

FROM（フロム）

```
FROM <イメージ>
```

または

```
FROM <イメージ>:<タグ>
```

または

```
FROM <イメージ>@<digest>
```

FROM 命令は、以降の命令で使うベース・イメージを指定します。あるいは、有効な Dockerfile は、1 行めを FROM 命令で指定する必要があります。イメージとは、あらゆる有効なものが利用できます。パブリック・リポジトリから**イメージを取得する**方法が一番簡単です。

- Dockerfile では、コメント以外では FROM を一番始めに書く必要があります。
- 単一の Dockerfile から複数のイメージを作成するため、複数の FROM を指定できます。各 FROM 命令ごとに自動的にコミットし、最新のイメージ ID が出力されるのを覚えておいてください。
- タグや digest 値はオプションです。省略した場合、ビルダーはデフォルトの latest とみなします。ビルダーは一致する tag 値が無ければエラーを返します。

MAINTAINER（メンテナ）

```
MAINTAINER <名前>
```

MAINTAINER 命令は、生成するイメージの Author（作者フィールドを指定します。

RUN（ラン）

RUN には２つの形式があります。

- RUN <コマンド>（シェル形式、コマンドをシェル /bin/sh -c で実行する）
- RUN ["実行バイナリ", "パラメータ１", "パラメータ２"]（exec 形式）

RUN 命令は既存イメージ上の新しいレイヤで、あらゆるコマンドを実行し、その結果をコミットする命令です。コミットの結果得られたイメージは、Dockerfile の次のステップで使われます。

RUN 命令の積み重ねとコミットによるイメージ生成は、Docker の中心となるコンセプト（概念）に従ったものです。コミットは簡単であり、ソース・コントロールのように、イメージの履歴上のあらゆる場所からコンテナを作成可能です。

exec 形式はシェルの文字列を変更できないようにします。また、 /bin/sh がベース・イメージに含まれなくても RUN コマンドを使えます。

シェル形式では、RUN 命令を \（バックスラッシュ）を使い、次の行と連結します。例えば、次の２行があるとします。

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

これは、次のように１行にできます。

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

「/bin/sh/」以外のシェルを使いたい場合は、exec 形式で任意のシェルを指定します。例： RUN ["/bin/bash", "-c", "echo hello"]。

exec 形式は JSON 配列でパースされます。つまり、文字を囲むのはシングル・クォート(') ではなくダブル・クォート(")を使う必要があります。

シェル形式と異なり、exec 形式はコマンド・シェルを呼び出しません。つまり、通常のシェルによる処理が行われません。例えば RUN ["echo", "$HOME"] は $HOME の変数展開を行いません。シェルによる処理を行いたい場合は、シェル形式を使うか、あるいはシェルを直接指定します。例：RUN ["sh", "-c", "echo", "$HOME"]。

次の構築時、RUN 命令によるキャッシュは自動的に無効化できません。RUN apt-get dist-upgrade -y のような命令のキャッシュがあれば、次の構築時に再利用されます。RUN 命令でキャッシュを使いたくない場合は、--no-cache フラグを使います。例： docker build --no-cache .

RUN 命令のキャッシュは、ADD 命令によって無効化されます。詳細は「ADD」命令のセクションをご覧ください。

既知の問題(RUN)

Issue 783[*1] は、AUFS ファイルシステム使用時、ファイルのパーミッションに関する問題が起こり得ます。例え

*1 https://github.com/docker/docker/issues/783

ば、ファイルを rm しようとする場合は注意が必要です。

最近の aufs バージョンを使っているシステムでは（例：dirperm1 マウント・オプションが利用可能 ）、docker は dirperm1 オプションのレイヤをマウント時、自動的に問題を修正しようとします。dirperm1 オプションに関する詳細は、aufs man ページ[1] をご覧ください。

システムが dirperm1 をサポートしていない場合は、issue に回避方法があります。

*1 http://aufs.sourceforge.net/aufs3/man.html

CMD

CMD には３つの形式があります。

- CMD ["実行バイナリ", "パラメータ１", "パラメータ２"] （exec 形式、推奨する形式）
- CMD ["パラメータ１", "パラメータ２"] （ENTRYPOINT のデフォルト・パラメータ）
- CMD <コマンド> （シェル形式）

Dockerfile で CMD 命令を一度だけ指定できます。複数の CMD がある場合、最も後ろの CMD のみ有効です。

CMD の主な目的は、**コンテナ実行時のデフォルトを提供します**。デフォルトには、実行可能なコマンドが含まれているか、あるいは省略されるかもしれません。省略時は ENTRYPOINT 命令で同様に指定する必要があります。

ENTRYPOINT 命令のデフォルトの引数として CMD を使う場合、CMD と ENTRYPOINT 命令の両方が JSON 配列フォーマットになっている必要があります。

exec 形式は JSON 配列でパースされます。つまり、文字を囲むのはシングル・クォート(') ではなくダブル・クォート(")を使う必要があります。

シェル形式と異なり、exec 形式はコマンド・シェルを呼び出しません。つまり、通常のシェルによる処理が行われません。例えば CMD ["echo", "$HOME"] は $HOME の変数展開を行いません。シェルによる処理を行いたい場合は、 シェル 形式を使うか、あるいはシェルを直接使います。例： CMD ["sh", "-c", "echo", "$HOME"] 。

シェルあるいは exec 形式を使う時、CMD 命令はイメージで実行するコマンドを指定します。

CMD をシェル形式を使えば、 <コマンド> は /bin/sh -c で実行されます。

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

<コマンド>をシェルを使わずに実行したい場合、コマンドを JSON 配列で記述し、実行可能なフルパスで指定する必要があります。**配列の形式が CMD では望ましい**形式です 。あらゆる追加パラメータは個々の配列の文字列として指定する必要があります。

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

もしコンテナで毎回同じものを実行するのであれば、CMD と ENTRYPOINT の使用を検討ください。詳細は ENTRYPOINT をご覧ください。

ユーザが docker run で引数を指定した時、これらは CMD で指定したデフォルトを上書きします。

RUN と CMD を混同しないでください。RUN が実際に行っているのは、コマンドの実行と結果のコミットです。一方の CMD は構築時には何もしませんが、イメージで実行するコマンドを指定します。

LABEL（ラベル）

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

LABEL 命令はイメージにメタデータを追加します。LABEL はキーとバリューのペアです。LABEL の値に空白スペースを含む場合はクォートを使いますし、コマンドラインの分割にバックスラッシュを使います。使用例：

```
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."
```

イメージは複数のラベルを持てます。複数のラベルを指定したら、 Docker は可能であれば 1 つの LABEL にすることをお勧めします。各 LABEL 命令は新しいレイヤを準備しますが、多くのラベルを使えば、それだけレイヤを使います。次の例は 1 つのイメージ・レイヤを使うものです。

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

上記の例は、次のようにも書き換えられます。

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

ラベルには、FROM イメージが使う LABEL も含まれています。ラベルのキーが既に存在している時、Docker は特定のキーを持つラベルの値を上書きします。

イメージが使っているラベルを確認するには、docker inspect コマンドを使います。

```
"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"
},
```

エクスポーズ
EXPOSE

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

EXPOSE 命令は、特定のネットワーク・ポートをコンテナが実行時にリッスンすることを Docker に伝えます。EXPOSE があっても、これだけではホストからコンテナにアクセスできるようにしません。アクセスするには、 -p フラグを使ってポートの公開範囲を指定するか、 -P フラグで全ての露出ポートを公開する必要があります。外部への公開時は他のポート番号も利用可能です。

ホストシステム上でポート転送を使うには、docker run リファレンスをご覧ください。Docker のネットワーク機能は、ネットワーク内でポートを公開しないネットワークを作成可能です。詳細な情報はユーザガイドをご覧ください。

ENV（エンブ）

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

ENV 命令は、環境変数 <key> と値 <value> のセットです。値は Dockerfile から派生する全てのコマンド環境で利用でき、インラインで置き換えも可能です。

ENV 命令は２つの形式があります。１つめは、ENV <key> <value> であり、変数に対して１つの値を設定します。はじめの空白以降の文字列が <value> に含まれます。ここには空白もクォートも含まれます。

２つめの形式は ENV <key>=<value> ... です。これは一度に複数の変数を指定できます。先ほどと違い、構文の２つめにイコールサイン（=）があるので気を付けてください。コマンドラインの分割、クォート、バックスラッシュは、空白スペースも含めて値になります。

例：

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

そして

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

この例では、どちらも最終的に同じ結果をコンテナにもたらしますが、私たちが推奨するのは前者です。理由は前者であれば単一のキャッシュ・レイヤしか使わないからです。

環境変数の設定に ENV を使えば、作成したイメージを使ってコンテナを実行しても有効です。どのような値が設定されているかは docker inspect で確認でき、変更するには docker run --env <key>=<value> を使います。

環境変数の一貫性は予期しない影響を与える場合があります。例えば、ENV DEBIAN_FRONTEND noninteractive が設定されていると、Debian ベースのイメージで apt-get の利用者が混乱するかもしれません。１つのコマンドだけで値を設定するには、RUN <key>=<value> <コマンド> を使います。

ADD（アッド）

Add は２つの形式があります。

- ADD <ソース>... <送信先>
- ADD ["<ソース>", ... "<送信先>"]（この形式はパスに空白スペースを使う場合に必要）

ADD 命令は <ソース> にある新しいファイルやディレクトリをコピー、あるいはリモートの URL からコピーします。それから、コンテナ内のファイルシステム上にある送信先に指定されたパスに追加します。

複数の <ソース> リソースを指定できます。この時、ファイルやディレクトリはソースディレクトリ（構築時のコンテクスト）からの相対パス上に存在しないと構築できません。

それぞれの <ソース> にはワイルドカードと Go 言語の filepath.Mach ルールに一致するパターンが使えます。例えば、次のような記述です。

```
ADD hom* /mydir/        # "hom" で始まる全てのファイルを追加
ADD hom?.txt /mydir/    # ? は１文字だけ一致します。例： "home.txt"
```

<送信先> は絶対パスです。あるいは、パスは WORKDIR からの相対パスです。ソースにあるものが、対象となる送信先コンテナの中にコピーされます。

```
ADD test relativeDir/          # "test" を `WORKDIR`/relativeDir/ （相対ディレクトリ）に追加
ADD test /absoluteDir/          # "test" を /absoluteDir/ （絶対ディレクトリ）に追加
```

追加される新しいファイルやディレクトリは、全て UID と GID が 0 として作成されます。

<ソース> がリモート URL の場合は、送信先のパーミッションは 600 にします。もしリモートのファイルが HTTP Last-Modified ヘッダを返す場合は、このヘッダの情報を元に送信先ファイルの mtime を指定するのに使います。しかしながら、ADD を使ったファイルをコピーする手順では、mtime はファイルが更新されたかどうかの決定には使われず、ファイルが更新されればキャッシュも更新されます。

Dockerfile を標準入力（ docker build - < 何らかのファイル ）を通して構築しようとしても。構築時のコンテントは存在しないため、Dockerfile には URL を指定する ADD 命令のみ記述可能です。また、圧縮ファイルを標準入力（ docker build - < archive.tar.gz ）を通すことができ、アーカイブに含まれるルートに Dockerfile があれば、構築時のコンテクストとしてアーカイブが使われます。

URL で指定したファイルに認証がかかっている場合は、RUN wget や RUN curl や他のツールを使う必要があります。これは ADD 命令が認証機能をサポートしていないからです。

ADD 命令の処理時、まず <ソース> に含まれる内容が変更されていれば、以降の Dockerfile に書かれている命令のキャッシュを全て無効化します。これは RUN 命令のキャッシュ無効化も含まれます。より詳細な情報については Dockerfile のベスト・プラクティス・ガイドをご覧ください。

ADD は以下のルールに従います。

- <ソース> パスは、構築時のコンテント内にある必要があります。そのため、ADD ../something /something の指定はできません。docker build の最初のステップで、コンテクストのディレクトリ（と、サブディレクトリ）を docker デーモンに送るためです。
- <ソース> が URL であり、 <送信先> の末尾にスラッシュが無い場合、URL からファイルをダウンロード

し、 <送信先> にコピーします。

- もし <ソース> が URL であり、 <送信先> の末尾がスラッシュの場合、URL からファイル名を推測し、ファイルを <送信先>/<ファイル名> にダウンロードします。例えば、ADD http://example.com/foobar / は、/foobar ファイルを作成します。URL には何らかのパスが必要です。これは適切なファイル名を見つけられない場合があるためです（今回の例では、http://example.com の指定は動作しません)。
- <ソース> がディレクトリの場合、ディレクトリの内容の全てをコピーします。これにはファイルシステムのメタデータを含みます。

ディレクトリ自身はコピーされません。ディレクトリは単なるコンテントの入れ物です。

- もし <ソース> がローカルにある tar アーカイブの場合、圧縮フォーマットを認識します（gzip、bzip2、xz を認識)。それからディレクトリに展開します。 リモートの URL が指定された場合は展開**しません**。ディレクトリにコピーまたは展開する時は、 tar -x と同じ働きをします。結果は次の処理を同時に行います。

1. 送信先のパスが存在しているかどうか
2. ファイル単位の原則に従って、ソース・ツリーの内容と衝突しないかどうか「2」を繰り返す

ファイルが圧縮フォーマットと認識するか、あるいはファイルの集まりをベースにしているのかは、ファイルの名前では判断しません。例えば、空のファイル名の拡張子が .tar.gz だとしても、圧縮ファイルと認識しないため、展開エラーのメッセージを表示**しません**。そして単純に送信先にファイルをコピーします。

- もし <ソース> がファイル以外であれば、個々のメタデータと一緒にコピーします。 <送信先> の末尾がスラッシュ / で終わる場合は、ディレクトリであるとみなし、 <ソース> の内容を <送信先>/base(<ソース>) に書き込みます。
- もし複数の <ソース> リソースが指定された場合や、ディレクトリやワイルドカードを使った場合、 <送信先> は必ずディレクトリになり、最後はスラッシュ / にしなければいけません。
- もし <送信先> の末尾がスラッシュで終わらなければ、通常のファイルとみなされ、 <ソース> の内容は <送信先> として書き込まれます。
- <送信先> が存在しなければ、パスに存在しないディレクトリを作成します。

COPY（コピー）

COPY は２つの形式があります。

- COPY <ソース>... <送信先>
- COPY ["<ソース>",... "<送信先>"]（この形式はパスに空白スペースを使う場合に必要）

COPY 命令は <ソース> にある新しいファイルやディレクトリをコピーするもので、コンテナ内のファイルシステム上にある <送信先> に指定されたパスに追加します。

複数の <ソース> リソースを指定できます。この時、ソースディレクトリ（構築時のコンテクスト）からの相対パス上に存在しないと構築できません。

それぞれの <ソース> にはワイルドカードと Go 言語の filepath.Mach ルールに一致するパターンが使えます。例えば、次のような記述です。

```
COPY hom* /mydir/        # "hom" で始まる全てのファイルを追加
COPY hom?.txt /mydir/    # ? は 1 文字だけ一致します。例： "home.txt"
```

<送信先> は絶対パスです。あるいは、パスは WORKDIR からの相対パスです。ソースにあるものが、対象となる送信先コンテナの中にコピーされます。

```
COPY test relativeDir/   # "test" を `WORKDIR`/relativeDir/ （相対ディレクトリ）に追加
COPY test /absoluteDir/  # "test" を /absoluteDir/ （絶対ディレクトリ）に追加
```

追加される新しいファイルやディレクトリは、全て UID と GID が 0 として作成されます。

標準入力（ docker build - < 何らかのファイル ）を使って構築しようとしても、構築時のコンテントは存在しないため、 COPY を使えません。

COPY は以下のルールに従います。

- <ソース> パスは、構築時のコンテント内にある必要があります。そのため、 COPY ../something /something の指定はできません。docker build の最初のステップで、コンテクストのディレクトリ（と、サブディレクトリ）を docker デーモンに送るためです。
- <ソース> がディレクトリの場合、ディレクトリ内容の全てをコピーします。これにはファイルシステムのメタデータを含みます。

ディレクトリ自身はコピーしません。ディレクトリは単なるコンテントの入れ物です。

- もし <ソース> がファイル以外であれば、個々のメタデータと一緒にコピーします。 <送信先> の末尾がスラッシュ / で終わる場合は、ディレクトリであるとみなし、 <ソース> の内容を <送信先>/base(<ソース>) に書き込みます。
- もし複数の <ソース> リソースが指定された場合や、ディレクトリやワイルドカードを使った場合、 <送信先> は必ずディレクトリになり、最後はスラッシュ / にしなければいけません。
- もし <送信先> の末尾がスラッシュで終わらなければ、通常のファイルとみなされ、 <ソース> の内容は <送信先> として書き込まれます。
- <送信先> が存在しなければ、パスに存在しないディレクトリを作成します。

ENTRYPOINT（エントリポイント）

ENTRYPOINT には２つの形式があります。

- ENTRYPOINT ["実行可能なもの", "パラメータ１", "パラメータ２"]（exec 形式、推奨）
- ENTRYPOINT コマンド パラメータ１ パラメータ２ （シェル形式）

ENTRYPOINT はコンテナが実行するファイルを設定します。
例えば、次の例は nginx をデフォルトの内容で開始し、ポート 80 を開きます。

```
docker run -i -t --rm -p 80:80 nginx
```

コマンドラインで docker run <イメージ> コマンドに引数を付けますと、exec 形式の ENTRYPOINT で指定した全要素の後に追加します。そして、この時に CMD を使って指定していた要素を上書きします。この動きにより、引数はエントリ・ポイント（訳者注：指定されたバイナリ）に渡されます。例えば、docker run <イメージ> -d は、引数 -d をエントリ・ポイントに渡します。ENTRYPOINT 命令を上書きするには、 docker run --entrypoint フラグを使います。

シェル形式では CMD や run コマンド行の引数を使えないという不利な点があります。ENTRYPOINT は /bin/sh -c のサブコマンドとして実行されるため、シグナルを渡せません。つまり、何かを実行してもコンテナの PID 1 にはなりません。そして、Unix シグナルを受け付けません。そのため、実行ファイルは docker stop <コンテナ> を実行しても、SIGTERM を受信しません。

なお、Dockerfile の最後に現れた ENTRYPOINT 命令のみ有効です。

exec 形式の ENTRYPOINT 例

ENTRYPOINT の exec 形式を使い、適切なデフォルトのコマンドと引数を指定します。それから CMD を使い、変更する可能性のある追加のデフォルト引数も指定します。

```
FROM ubuntu
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
```

コンテナを実行したら、top のプロセスが１つだけ見えます。

```
$ docker run -it --rm --name test  top -H
top - 08:25:00 up  7:27,  0 users,  load average: 0.00, 0.01, 0.05
Threads:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   2056668 total,  1616832 used,   439836 free,    99352 buffers
KiB Swap:  1441840 total,        0 used,  1441840 free.  1324440 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
    1 root      20   0   19744   2336   2080 R  0.0  0.1   0:00.04 top
```

より詳細なテストをするには、docker exec コマンドが使えます。

```
$ docker exec -it test ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  2.6  0.1  19752  2352 ?        Ss+  08:24   0:00 top -b -H
root         7  0.0  0.1  15572  2164 ?        R+   08:25   0:00 ps aux
```

それから、docker stop test を使い top を停止するよう、通常のリクエストを行えます。
次の Dockerfile は ENTRYPOINT を使って Apache をフォアグラウンドで実行します（つまり、PID 1 として）。

```
FROM debian:stable
RUN apt-get update && apt-get install -y --force-yes apache2
EXPOSE 80 443
VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

もし実行するだけの起動スクリプトを書く必要があれば、最後に実行するコマンドが Unix シグナルを受信できるよう、exec と gosu コマンドを使うことで可能になります。

```
#!/bin/bash
set -e

if [ "$1" = 'postgres' ]; then
    chown -R postgres "$PGDATA"

    if [ -z "$(ls -A "$PGDATA")" ]; then
        gosu postgres initdb
    fi

    exec gosu postgres "$@"
fi

exec "$@"
```

もしも、シャットダウン時に何らかの追加クリーンアップ（あるいは、他のコンテナとの通信）が必要な場合や、1 つ以上の実行ファイルと連携したい場合は、ENTRYPOINT のスクリプトが Unix シグナルを受信できるようにし、それを使って様々な処理を行います。

```
#!/bin/sh
# メモ：これは sh を使っていますので、busyboy コンテナでも動きます

# サービス停止時に手動でもクリーンアップが必要な場合は trap を使います。
# あるいは 1 つのコンテナ内に複数のサービスを起動する必要があります。
trap "echo TRAPed signal" HUP INT QUIT KILL TERM

# ここからバックグラウンドでサービスを開始します
/usr/sbin/apachectl start

echo "[hit enter key to exit] or run 'docker stop <container>'"
read

# ここからサービスを停止し、クリーンアップします
echo "stopping apache"
/usr/sbin/apachectl stop

echo "exited $0"
```

このイメージを docker run -it --rm -p 80:80 --name test apache で実行したら、コンテナのプロセス状態を docker exec や docker top で調べられます。それから、スクリプトに Apache 停止を依頼します。

```
$ docker exec -it test ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.1  0.0   4448   692 ?        Ss+  00:42   0:00 /bin/sh /run.sh 123 cmd cmd2
root         19  0.0  0.2  71304  4440 ?        Ss   00:42   0:00 /usr/sbin/apache2 -k start
www-data     20  0.2  0.2 360468  6004 ?        Sl   00:42   0:00 /usr/sbin/apache2 -k start
www-data     21  0.2  0.2 360468  6000 ?        Sl   00:42   0:00 /usr/sbin/apache2 -k start
root         81  0.0  0.1  15572  2140 ?        R+   00:44   0:00 ps aux
$ docker top test
PID                 USER                COMMAND
10035               root                {run.sh} /bin/sh /run.sh 123 cmd cmd2
10054               root                /usr/sbin/apache2 -k start
10055               33                  /usr/sbin/apache2 -k start
10056               33                  /usr/sbin/apache2 -k start
$ /usr/bin/time docker stop test
test
real 0m 0.27s
user 0m 0.03s
sys  0m 0.03s
```

ENTRYPIONT 設定は --entrypoint を使って上書きできますが、設定できるのはバイナリが実行可能な場合のみです（ sh -c が使われていない時のみ ）。

exec 形式は JSON 配列でパースされます。つまり、語句はシングルクォート(')ではなく、ダブルクォート(")で囲む必要があります。

シェル 形式とは異なり、exec 形式はシェルを呼び出しません。つまり、通常のシェル上の処理はされません。例えば、ENTRYPOINT ["echo", "$HOME"] は $HOME を変数展開しません。シェル上の処理が必要であれば、 シェル 形式を使うか、シェルを直接実行します。例： ENTRYPOINT ["sh", "-c", "echo", "$HOME"]。変数は Dockerfile で ENV を使って定義することができ、Dockerfile パーサー上で展開されます。

シェル形式の ENTRYPOINT 例

ENTRYPOINT に文字列を指定したら、 /bin/sh -c で実行されます。この形式はシェルの処理を使いますので、シェル上の環境変数を展開し、CMD や docker run コマンド行の引数を無視します。docker stop で ENTRYPOINT で指定している実行ファイルにシグナルを送りたい場合は、exec を使う必要があるのを思い出してください。

```
FROM ubuntu
ENTRYPOINT exec top -b
```

このイメージを実行したら、単一の PID 1 プロセスが表示されます。

```
$ docker run -it --rm --name test top
Mem: 1704520K used, 352148K free, 0K shrd, 0K buff, 140368121167873K cached
CPU:   5% usr   0% sys   0% nic  94% idle   0% io   0% irq   0% sirq
Load average: 0.08 0.03 0.05 2/98 6
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
    1     0 root     R     3164   0%   0% top -b
```

終了するには、docker stop を実行します。

```
$ /usr/bin/time docker stop test
test
real    0m 0.20s
user    0m 0.02s
sys 0m 0.04s
```

ENTRYPOINT に exec を追加し忘れたとします。

```
FROM ubuntu
ENTRYPOINT top -b
CMD --ignored-param1
```

次のように実行します（次のステップで名前を使います)。

```
$ docker run -it --name test top --ignored-param2
Mem: 1704184K used, 352484K free, 0K shrd, 0K buff, 140621524238337K cached
CPU:   9% usr   2% sys   0% nic  88% idle   0% io   0% irq   0% sirq
Load average: 0.01 0.02 0.05 2/101 7
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
    1     0 root     S     3168   0%   0% /bin/sh -c top -b cmd cmd2
    7     1 root     R     3164   0%   0% top -b
```

top の出力から、ENTRYPOINT が PID 1 ではないことが分かるでしょう。
それから docker stop test を実行しても、コンテナはすぐに終了しません。これは stop コマンドがタイムアウト後、SIGKILL を強制送信したからです。

```
$ docker exec -it test ps aux
PID   USER     COMMAND
    1 root     /bin/sh -c top -b cmd cmd2
    7 root     top -b
    8 root     ps aux
$ /usr/bin/time docker stop test
test
real    0m 10.19s
user    0m 0.04s
sys 0m 0.03s
```

CMD と ENTRYPOINT がどのように作用するか学ぶ

CMD と ENTRYPOINT 命令はコンテナ起動時に実行するコマンドを定義します。両方を記述する時、動作には複数のルールがあります。

1. Dockerfile には少なくとも 1 つの CMD または ENTRYPOINT 命令を含むべきです。

2. ENTRYPOINT は実行可能なコンテナとして定義する時に使うべきです。

3. コンテナをアドホック（その場その場）で実行するコマンドを ENTRYPOINT にする場合、そのデフォルトの引数の指定として CMD を指定すべきです。

4. CMD はコンテナ実行時に引数を指定すると上書きします。

以下の表は ENTRYPOINT / CMD を組み合わせたコマンドの実行結果です。

	ENTRYPOINT なし	ENTRYPOINT exec_entry p1_entry	NTRYPOINT ["exec_entry", "p1_entry"]
CMD なし	エラー。実行できない	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD ["exec_cmd","p1_cmd"]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry exec_cmd p1_cmd	exec_entry p1_entry exec_cmd p1_cmd
CMD ["p1_cmd", "p2_cmd"]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry p1_cmd p2_cmd	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

VOLUME（ボリューム）

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

VOLUME 命令は指定した名前でマウントポイントを作成し、他のホストやコンテナから外部マウント可能なボリュームにします。指定する値は `VOLUME ["/var/log"]` といった JSON 配列になるべきです。あるいは文字列で `VOLUME /var/log` や `VOLUME /var/log /var/db` のように、複数の引数を書くこともできます。Docker クライアントを使ったマウント命令や詳しい情報やサンプルは「ボリュームを経由してディレクトリを共有」のセクションをご覧ください。

`docker run` コマンドは、ベース・イメージから指定した場所に、データを保存する場所として新規作成したボリュームを初期化します。例えば、次の Dockerfile をご覧ください。

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

この Dockerfile によって作られたイメージは、`docker run` を実行したら、新しいマウント・ポイント `/myvol` を作成し、`greeting` ファイルを直近で作成したボリュームにコピーします。

構築ステップでボリューム内においてあらゆる変更を加えても、宣言後に内容は破棄されます。

リストは JSON 配列でパースされます。これが意味するのは、単語はシングルクォート(')で囲むのではなく、ダブルクォート(")を使う必要があります。

USER（ユーザ）

```
USER daemon
```

USER 命令セットはユーザ名か UID を使います。これはイメージを RUN、CMD、ENTRYPOINT 命令で実行時のものであり、Dockerfile で指定します。

ワークディレクトリ
WORKDIR

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

WORKDIR 命令セットは Dockerfile で RUN、CMD、ENTRYPOINT、COPY、ADD 命令実行時の作業ディレクトリ（working directory）を指定します。もし WORKDIR が存在しなければ、Dockerfile 命令内で使用しなくてもディレクトリを作成します。

1 つの Dockerfile で複数回の利用が可能です。パスを指定したら、WORKDIR 命令は直前に指定した相対パスに切り替えます。例：

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

この Dockerfile を使えば、最後の pwd コマンドの出力は /a/b/c になります。

WORKDIR 命令は ENV 命令を使った環境変数も展開できます。環境変数を使うには Dockerfile で明確に定義する必要があります。例：

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

この Dockerfile を使えば、最後の pwd コマンドの出力は /path/$DIRNAME になります。

ARG（アーグ）

```
ARG <名前>[=<デフォルトの値>]
```

ARG 命令は、構築時に作業者が docker build コマンドで使う変数、 --build-arg <変数名>=<値> フラグを定義するものです。ユーザが構築時に引数を指定しても Dockerfile で定義されていなければ、構築時に次のようなエラーが出ます。

```
One or more build-args were not consumed, failing build.
```

Dockerfile の作者は ARG 変数を 1 度だけ定義するだけでなく、複数の ARG を指定可能です。有効な Dockerfile の例：

```
FROM busybox
ARG user1
ARG buildno
...
```

Dockerfile の作者は、オプションで ARG 命令のデフォルト値を指定できます。

```
FROM busybox
ARG user1=someuser
ARG buildno=1
...
```

ARG がデフォルト値を持っている場合、構築時に値の指定が無ければ、このデフォルト値を使います。
ARG 変数は Dockerfile で記述した行以降で効果があります。ただし、コマンドライン上で引数の指定が無い場合です。次の Dockerfile の例を見てみましょう。

```
FROM busybox
USER ${user:-some_user}
ARG user
USER $user
...

$ docker build --build-arg user=what_user Dockerfile
```

２行の USER は some_user を、３行めサブシーケントで定義された user 変数として評価します。４行めでは what_user を USER で定義したものと評価し、what_user 値はコマンドラインで指定したものになります。ARG 命令で定義するまで、あらゆる変数は空の文字列です。

構築時の変数として、GitHub の鍵やユーザの証明書などの秘密情報を含むのは、推奨される使い方ではありません。

ARG や ENV 命令を RUN 命令のための環境変数にも利用できます。ENV 命令を使った環境変数の定義は、常に同じ名前の ARG 命令を上書きします。Dockerfile における ENV と ARG 命令を考えましょう。

```
FROM ubuntu
ARG CONT_IMG_VER
ENV CONT_IMG_VER v1.0.0
RUN echo $CONT_IMG_VER
```

それから、イメージを次のように起動します。

```
$ docker build --build-arg CONT_IMG_VER=v2.0.1 Dockerfile
```

この例では、RUN 命令は v1.0.0 の代わりに、ARG でユーザから渡された v2.0.1 を使います。この動作はシェルスクリプトの挙動に似ています。ローカルのスコープにある環境変数が、与えられた引数や上位の環境変数によって上書きするようなものです。

上記の ENV 指定の他にも、更に ARG と ENV を使いやすくする指定も可能です。

```
FROM ubuntu
ARG CONT_IMG_VER
ENV CONT_IMG_VER ${CONT_IMG_VER:-v1.0.0}
RUN echo $CONT_IMG_VER
```

ARG 命令とは異なり、構築時の ENV 値は常に一定です。docker build で --build-arg フラグを使わない場合を考えてみましょう。

```
$ docker build Dockerfile
```

この Dockerfile の例では、CONT_IMG_VER はイメージの中では変わりませんが、3 行めの ENV 命令でデフォルト値を設定することにより、値は v1.0.0 となります。

この例における変数展開のテクニックは、コマンドラインから引数を渡せるようにし、ENV 命令を使うことで最終的に一貫したイメージを作成します。サポートされている変数展開は Dockerfile 命令の一部のみです。

Docker は Dockerfile に対応する ARG 命令が無くても、既定の ARG 変数セットを持っています。

- HTTP_PROXY
- http_proxy
- HTTPS_PROXY
- https_proxy
- FTP_PROXY
- ftp_proxy
- NO_PROXY
- no_proxy

これらを使うには、コマンドラインで --build-arg <変数名>=<値> フラグを単に渡すだけです。

構築キャッシュの影響

ARG 変数は、イメージ構築時の ENV 変数のように残り続けません。しかし、ARG 変数は構築キャッシュで似たような方法として扱えます。もし Dockerfile で ARG 変数を定義したら、この値が以前の値と違う時は、以降で ARG 変数が出た時「キャッシュ・ミス」を発生します。

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 RUN echo $CONT_IMG_VER
```

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 RUN echo hello
```

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 ENV CONT_IMG_VER $CONT_IMG_VER
4 RUN echo $CONT_IMG_VER
```

この例では、キャッシュミスが３行めで発生します。ミスが起こるのは ENV 変数が ARG 変数を参照しているのと、この変数がコマンドラインで変わるためです。例における ENV コマンドはイメージの中で処理されるものです。

もし ENV 命令を同じ名前の ARG 命令で、次のように上書きしたらどうでしょう。

```
1 FROM ubuntu
2 ARG CONT_IMG_VER
3 ENV CONT_IMG_VER hello
4 RUN echo $CONT_IMG_VER
```

３行めはキャッシュミスを引き起こしません。CONT_IMG_VAR は固定（hello）だからです。そのため、環境変数と値は RUN（４行め）で使われますが、構築時に変わりません。

ONBUILD（オンビルド）

ONBUILD [命令]

イメージは他で構築したイメージを元にしている時、ONBUILD 命令はイメージに対して最終的に実行するトリガ命令を追加します。トリガは構築後に行うもので、Dockerfile で FROM 命令のあとに書くことができます。

あらゆる構築時の命令をトリガとして登録可能です。

これは他のイメージからイメージを構築する時に役立つでしょう。例えば、アプリケーションの開発環境やデーモンは、ユーザごとに設定をカスタマイズする可能性があります。

例えば、イメージが Python アプリケーション・ビルダーを再利用する時、アプリケーションのソースコードを適切なディレクトリに追加し、その後、構築スクリプトを実行することもあるでしょう。この時点では ADD と RUN を呼び出せません。なぜなら、まだアプリケーションのソースコードにアクセスしておらず、個々のアプリケーション構築によって異なるからです。アプリケーションの開発者は、ボイラープレートである Dockerfile をコピーペーストでアプリケーションを入れるように編集するだけです。ですが、これは効率的ではなく、エラーを引き起こしやすく、アプリケーション固有のコードが混在することで更新が大変になります。

この解決方法として、ONBLUID を使い、実行後に別の構築ステージに進む上位命令を登録することです。

これは次のように動作します。

1. ONBUILD 命令が呼び出されたら、ビルダーはイメージ構築時のメタデータの中にトリガを追加します。

2. 構築が完了したら、全てのトリガはイメージのマニフェスト内の OnBuild キー配下に保管されます。この構築時点では、命令は何ら影響を与えません。

3. このイメージは後で何らかのイメージの元になります。その時は FROM 命令で呼び出されます。FROM 命令の処理の一部として、ダウンストリームのビルダーは ONBULID トリガを探し、登録された順番で実行します。もしトリガが失敗したら、FROM 命令は処理を中断し、ビルドを失敗とします。もし全てのトリガが成功したら、FROM 命令は完了し、以降は通常の構築が進みます。

4. 実行する前に、最終的なイメージ上からトリガが削除されます。言い替えると構築された「孫」には、何ら親子関係がありません。

次のような例の記述を追加するでしょう。

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

【警告】ONBUILD ONBUILD 命令を使って ONBULID 命令の上書きはできません。

STOPSIGNAL（ストップシグナル）

STOPSIGNAL シグナル

STOPSIGNAL 命令は、コンテナを終了する時に送信するための、システム・コール・シグナルを設定します。シグナルはカーネルの syscall テーブルと一致する、有効な番号の必要があります。例えば、9 あるいはシグナル名 SIGNAME や、SIGKILL などです。

Dockerfile の例

以下で Dockerfile 構文の例を参照できます。実際の環境に興味があれば、 Docker 化の例のセクションをご覧ください。

例:Nginx

```
# Nginx
#
# VERSION               0.0.1

FROM      ubuntu
MAINTAINER Victor Vieux <victor@docker.com>

LABEL Description="This image is used to start the foobar executable" Vendor="ACME Products"
Version="1.0"
RUN apt-get update && apt-get install -y inotify-tools nginx apache2 openssh-server
```

例:Firefox over VNC

```
# Firefox over VNC
#
# VERSION               0.3

FROM ubuntu

# 「フェイク」(偽) のディスプレイ用の vnc, xvfb と firefox をインストール
RUN apt-get update && apt-get install -y x11vnc xvfb firefox
RUN mkdir ~/.vnc
# パスワードをセットアップ
RUN x11vnc -storepasswd 1234 ~/.vnc/passwd
# firefox の自動起動 (ベストな方法ではありませんが、動きます)
RUN bash -c 'echo "firefox" >> /.bashrc'

EXPOSE 5900
CMD    ["x11vnc", "-forever", "-usepw", "-create"]
```

例:複数のイメージ

```
# 複数のイメージ例
#
# VERSION               0.1

FROM ubuntu
RUN echo foo > bar
# 「===> 907ad6c2736f」 のような出力があります

FROM ubuntu
RUN echo moo > oink
# 「===> 695d7793cbe4」 のような出力があります

# You ' ll now have two images, 907ad6c2736f with /bar, and 695d7793cbe4 with
# /oink.
# これで2つのイメージができました。
# /bar がある 907ad6c2736f と、/oink がある 695d7793cbe4 です
```