はじめに

DockerEngineユーザガイド〜基礎編

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 12 日　beta1 を公開

目次

1章

1.1 Docker Engine について

Develop, Ship and Run Any Application, Anywhere

あらゆるアプリケーションを、どこでも開発、移動、実行できるように

Docker[*1] とは、開発者やシステム管理者が、アプリケーションの開発、移動、実行するためのプラットフォームです。**Docker**（ドッカー） は部品（コンポーネント）から迅速にアプリケーションを組み立てるため、コードの移動による摩擦を無くします。Docker はコードのテストやプロダクション（本番環境）に対する迅速な展開をもたらします。

Docker を構成するのは次の２つです。

- **Docker Engine**（ドッカーエンジン） … 私たちの軽量かつ強力なオープンソースによるコンテナ仮装化技術であり、アプリケーションの構築からコンテナ化に至るワークフローを連結。
- **Docker Hub**（ドッカーハブ） … 皆さんのアプリケーション群を共有・管理する SaaS[*2] サービス。

1.1.1 なぜ Docker なのでしょうか

より速いアプリケーションの配信

私たちは皆さんの環境を良くしたいのです。Docker コンテナと、そのワークフローにより、開発者、システム管理者、品質管理担当者、リリース・エンジニアが一緒になり、皆さんのコードをプロダクションに運ぶのを手伝い、使いやすくします。私たちが作成した標準コンテナ形式により、開発者はコンテナの中のアプリケーションに集中し、システム管理者やオペレータはコンテナのデプロイと実行が可能になります。この作業範囲の分割によって、コードの開発と管理を単純化します。

*1 Docker の公式サイトは https://www.docker.com/

*2 SaaS = Software as a Service の略で、インターネットを通して提供するサービスの意味。

私たちは新しいコンテナを簡単に構築できるようにしました。これにより、アプリケーションの迅速な逐次投入や、変更の視認性を高めます。この機能は、皆さんの組織における誰もが、アプリケーションをどのように構築し、どのように動作するのかを理解する手助けとなるでしょう。

Docker コンテナは軽量かつ高速です！ コンテナの起動時間は数秒であり、開発・テスト・デプロイのサイクルにかかる時間を減らします。

デプロイやスケールをもっと簡単に

Docker コンテナは（ほとんど）どこでも動きます。コンテナのデプロイは、デスクトップでも、物理サーバでも、仮想マシンにもデプロイできます。それだけでなく、あらゆるデータセンタ、パブリック・クラウド、プライベート・クラウドにデプロイできます。

Docker は多くのプラットフォームで動作しますので、アプリケーション周辺の移動も簡単です。必要であればテスト環境上のアプリケーションを、クラウド環境でもどこでも簡単に移動できます。

また、Docker の軽量コンテナは、スケールアップやスケールダウンを速く簡単にします。必要であれば迅速に多くのコンテナを起動できますし、必要がなくなれば簡単に停止できます。

より高い密度で多くの仕事量を

Docker コンテナはハイパーバイザーが不要なため、ホスト上により多くを集約できます。つまり、各サーバの価値をより高め、機材やライセンスの消費を減らせる可能性があります。

デプロイの高速化による管理の簡易化

Docker がもたらすワークフローの高速化は、小さな変更だけでなく、大規模アップデートに至るまでをも簡単にします。小さな変更とは、更新時におけるリスク（危険性）の減少を意味します。

1.1.2 このガイドについて

2章「Docker のアーキテクチャ」は以下の理解を助けます。

- Docker がハイレベルでどのように動作するのか
- Docker アーキテクチャの理解
- Docker の機能確認
- Docker と仮想化の違いを知れる
- 一般的な使い方を知れる

インストールガイド

3章「インストール」は、様々なプラットフォームに対する Docker のインストール方法を理解します。

Docker ユーザガイド

Docker の詳細を学び、使い方や実装に関する疑問を解消するには、 5章「Docker Engine ユーザガイド」をご確認ください。

1.1.3 リリースノート

各リリースにおける変更点の概要については、 リリース・ノートの各ページ[1] をご確認ください。

*1 https://docs.docker.com/release-notes

1.1.4 機能廃止ポリシー

Docker の各バージョンにおいて、既存機能の削除や、新しい機能に置き換わるような変更が生じる可能性があります。既存の機能を削除する前に、ドキュメントの中で「deprecated」（廃止予定）とラベル付けするようにします。通常、少なくとも２つのリリースがされるまで残し、その後、削除します。

利用者は、廃止予定の機能に関しては、リリースごとに注意をお払いください。機能の変更が分かった場合は、可能な限り速く（適切な）移行をお願いします。

廃止機能の一覧リストについては、廃止機能のページ[*1] をご覧ください。

1.1.5 使用許諾

Docker の使用許諾（ライセンス）は Apache License, Version 2.0 です。使用許諾条項の詳細は LICENSE[*2] をご覧ください。

*1 http://docs.docker.jp/engine/deprecated.html

*2 https://github.com/docker/docker/blob/master/LICENSE

2 章

アーキテクチャの理解

2.1 Docker とは

Docker はアプリケーションを開発（developing）・移動（shipping）・実行（running）するための、オープンなプラットフォームです。Docker は皆さんのアプリケーションをより速く運ぶために設計されています。Docker を使うことで、アプリケーションをインフラから分離し、アプリケーションを管理するようにインフラを扱えるようにします。Docker はコードの移動をより速く、テストを速く、デプロイを速くし、コードの記述とコードの実行におけるサイクルを短くします。

Docker はこれを実現するために、カーネルの**コンテナ化（containerization）機能**がもたらすワークフローと手法（ツール）を組みあわせます。それゆえ、アプリケーション管理とデプロイの手助けになるでしょう。

中心となるのは、あらゆるアプリケーションをコンテナ内で安全に**分離（isolated）**[*1] して実行する手法を、Docker が提供することです。分離とセキュリティにより、ホスト上で擬似的に多くのコンテナを実行できます。コンテナは軽量な性質のため、実行するのにハイパーバイザーのような外部装置を必要としません。つまり、ハードウェアに依存しないのです。

コンテナを取りまく手法（ツール）とプラットフォームは、様々な場所で役立つでしょう。

- アプリケーション（と、必要なコンポーネントを）を Docker コンテナの中に入れる
- 更に開発やテストのために、これらのコンテナをチームに配布・移動する
- プロダクション環境にアプリケーションをデプロイ

2.1.1 何のために Docker を使うのでしょうか

アプリケーションの速いデリバリ

Docker は 開発のライフサイクルの手助けに最適です。Docker は開発者がローカルのコンテナで開発できるようにし、そこにアプリケーションとサービスを入れられます。そして、継続的インテグレーションや、デプロイのワークフローと統合できます。

*1 isolate には「分離する」「隔離する」「独立する」という意味があり、他と隔たる状態を示します。

例えば、開発者がローカルでコードを書き、Docker 上の開発スタックを同僚と共有します。準備が整えば、コードとスタックを開発環境からテスト環境に移動し、開発環境のスタックをテスト環境に移動し、必要なテストを実行します。テスト環境のあとで、Docker イメージをプロダクション環境に送信し、コードをデプロイできるのです。

デプロイとスケールをより簡単に

Docker のコンテナを基盤としたプラットフォームは、ワークロードの高い可用性をもたらします。Docker コンテナは開発者のローカルホスト上で実行できるだけでなく、データセンタの物理環境や仮想マシン上や、クラウド上でも実行できます。

また、Docker のポータビリティと軽量な性質によって、動的なワークロードの管理を簡単にします。Docker を使えば、アプリケーションやサービスのスケールアップやティアダウンを簡単に行います。Docker のスピードが意味するのは、スケールをほぼリアルタイムに近く行えることです。

高密度と更にワークロードの実行を実現

Docker は軽量かつ高速です。これはハイパーバイザーをベースとした仮想化マシンよりも、費用対効果を高くします。これが特に使いやすいのは高密度の環境でしょう。例えば、自分たちのクラウドや PaaS（プラットフォーム・アズ・ア・サービス）においてです。しかし、自分たちが持っているリソースを、より活用したいとする中小規模のデプロイにも便利です。

2.1.2 Docker の主な構成要素は？

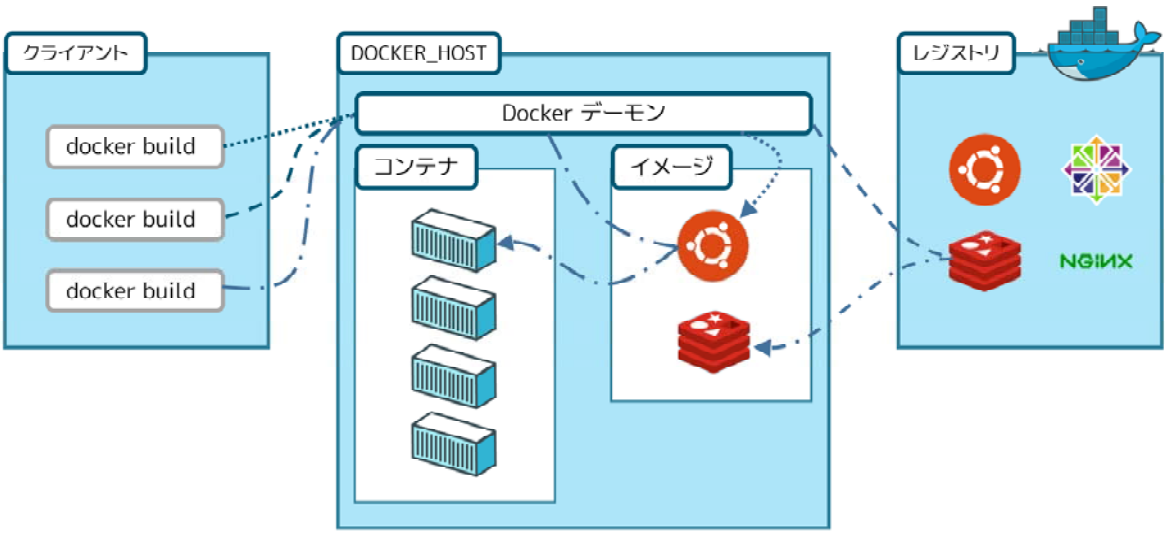
Docker は２つの主要コンポーネントを持ちます。

- Docker：オープンソースのコンテナ仮想化プラットフォーム
- Docker Hub：Docker コンテナを共有・管理する私たちの SaaS プラットフォームです。

Docker はオープンソースの Apache2.0 利用許諾（ライセンス）の下で提供されています。

2.1.3 Docker のアーキテクチャとは

Docker はクライアント・サーバ型のアーキテクチャです。**Docker クライアント**が Docker コンテナの構築・実行・配布といった力仕事をするには、**Docker デーモン**と通信します。 Docker クライアントとデーモンは、どちらも同じシステム上でも実行できます。あるいは、Docker クライアントはリモートの Docker デーモンに接続するのも可能です。Docker クライアントとデーモンは、お互いにソケットもしくは RESTful（レストフル） API（エーピーアイ） を経由して通信します。



Docker デーモン

上図で見たように、Docker デーモンはホストマシン上で動きます。ユーザは直接デーモンと通信せず、Docker クライアントを通して行います。

Docker クライアント

Docker クライアントは **docker バイナリ**の形式です。これは主にユーザが Docker との通信に使います。ユーザからのコマンドを受け付けたら、その先にある Docker デーモンが通信を返します。

Docker の内部

Docker 内部を理解するには、３つのコンポーネントを知れる必要があります。

- Docker イメージ（image）
- Docker レジストリ（registry）
- Docker コンテナ（container）

Docker イメージ

Docker イメージとは、読み込み専用（read-only）のテンプレートです。例えば、あるイメージは Ubuntu オペレーティング・システム上に、Apache とウェブ・アプリケーションが含まれるでしょう。イメージは Docker コンテナの作成時に使います。Docker は新しいイメージの構築や、既存イメージを更新します。あるいは、他の人が既に作成した Docker イメージをダウンロードすることも可能です。Docker イメージとは Docker における 構築（build）コンポーネントです。

Docker レジストリ

Docker レジストリはイメージを保管します。パブリックもしくはプライベートに保管するイメージの、アップロードやダウンロードが可能です。パブリックな Docker レジストリとしては、Docker Hub を提供されています。そこでは利用可能なイメージがたくさん提供されています。イメージを自分自身で作れるだけでなく、他人が作成したイメージも利用できます。Docker レジストリとは Docker における 配布（distribution） コンポーネントです。

Docker コンテナ

Docker コンテナはディレクトリと似ています。Docker コンテナはアプリケーションの実行に必要な全てを含みます。各コンテナは Docker イメージによって作られます。Docker コンテナは実行・開始・停止・移動・削除できます。各コンテナは分離されており、安全なアプリケーションのプラットフォームです。Docker コンテナとは Docker における実行（run）コンポーネントです。

2.1.4 どのようにして Docker は動作しますか

これまでに、次のことを学びました。

- アプリケーションを保持する Docker イメージを構築できる
- これらの Docker イメージでアプリケーションを実行する Docker コンテナを作成できる
- これらの Docker イメージを Docker Hub や自分のレジストリで共有できる

それでは、Docke が動作するために、それぞれの要素をどのように連携させているのか理解しましょう。

Docker イメージの役割

これまで分かったのは、Docker イメージとは読み込み専用のテンプレートであり、これを使って Docker コンテナを起動します。各イメージは**レイヤ（layer）**[*1] の積み重ねで構成されています。Docker はユニオン・ファイルシステム[*2]（UnionFS）を使い、これらのレイヤを単一のイメージに連結します。ユニオン・ファイルシステムは、ブランチとしても知れられています。これは透過的な重ね合わせ（overlaid）と、互いに密着した（coherent）ファイルシステムを形成します。

Docker が軽量な理由の 1 つが、これらのレイヤによるものです。Docker イメージに変更を加えたとしましょう。例えば、アプリケーションを新しいバージョンに更新すると仮定します。この更新時に新しいレイヤを構築します。つまり、仮想マシン上で何らかの作業をした結果、イメージの入れ替えや完全な再構築ではなく、単純にレイヤが追加するか更新するだけなのです。この新しいイメージの、配布に関する心配は不要です。新しい Docker イメージを速く簡単に配布するには、単に更新されたレイヤを配布するだけです。

各イメージは**ベース・イメージ（base image）**から作られます。例えば、 `ubuntu` は ベース Ubuntu イメージですし、 `fedora` はベース Fedora イメージです。また、自分自身で新しいイメージの元も作れます。例えば、自分でベース Apache イメージを作れば、これを自分用のウェブ・アプリケーション・イメージのベース（基礎）として使えます。

Docker は常にこれらのベース・イメージを Docker Hub から取得します。

これらのベース・イメージからシンプルに構築できるようにするため、Docker イメージには**命令**（instructions）と呼ぶ構築手順を簡単に記述した集まりがあります。それぞれの命令ごとに、イメージ上に新しいレイヤを作成します。命令は次のような動作をします。

*1 layer = 層、の意味です。

*2 Union File System https://ja.wikipedia.org/wiki/UnionFS

- コマンドの実行
- ファイルやディレクトリの追加
- 環境変数の作成
- 対象イメージを使ってコンテナを起動する時、どのプロセスを実行するか

これらの命令を **Dockerfile**（ドッカーファイル）と呼ぶファイルに保管します。Docker にイメージの構築を要求したら、Docker はこの Dockerfile を読み込み、命令を実行し、最終的なイメージを返します。

どのように Docker レジストリは動作するのか

Docker レジストリは Docker イメージを保管します。Docker イメージを構築後、Docker Hub のような公開レジストリに**送信**（**push**（プッシュ））します。あるいはファイアウォール背後にある自分のレジストリにも送信できます。

Docker クライアントを使い、公開済みのイメージを検索できます。そして、自分の Docker ホスト上にイメージを**取得**（**pull**（プル））、つまりダウンロードし、これを使ってコンテナを構築できます。

Docker Hub はイメージを保管するために、パブリックとプライベートなストレージの利用をサポートしています。パブリック・ストレージとは誰でも検索可能でダウンロードできるものです。プライベート・ストレージとは検索結果から除外され、自分もしくは許可されたユーザだけがイメージを取得し、コンテナを構築できるようにします。 ストレージの料金プランと契約はサイト上[*1] で行えます。

どのようにコンテナは動作するのか

コンテナに含まれているのは、オペレーティング・システム、ユーザが追加したファイル、メタデータです。これまで見てきたように、各コンテナはイメージから構築します。そのイメージは、Docker に対してどのコンテナの中に何があるか、コンテナ起動時に何のプロセスを実行するか、その他のデータに関する設定確認をします。Docker イメージは読み込み専用です。Docker がイメージからコンテナを実行する時、読み書き可能なレイヤを既存イメージ上に追加し（先ほど見た通り、ユニオン・ファイルシステムを使います）、アプリケーションを実行できるようにします。

コンテナを実行すると何が起きるのか

`docker` バイナリまたは API を経由して、Docker クライアントは Docker デーモンにコンテナ実行を命令します。

```
$ docker run -i -t ubuntu /bin/bash
```

このコマンドを分解（ブレイクダウン）してみましょう。Docker クライアントは `docker` バイナリを使って実行され、`run` オプションは新しいコンテナの起動を命令します。Docker クライアントが Docker デーモンに対してコンテナを起動する時、最低限必要なのは以下の項目です。

- コンテナを何の Docker イメージで構築するのか。ここでは `ubuntu` のベース Ubuntu イメージを使用
- コンテナを起動したら、その中で何のコマンドを実行したいのか、ここでは `/bin/bash` を指定し、新しいコンテナの中で Bash シェルを開始

それでは、このコマンドの水面下では何が起こっているのでしょうか。

Docker の処理内容を、順番に見ていきます。

*1 https://hub.docker.com/plans

- ubuntu イメージの取得：Docker は `ubuntu` イメージの存在を確認し、もしローカルホスト上に存在しなければ、Docker Hub からダウンロードする。イメージが既にあれば、Docker はこれを新しいコンテナのために使う。
- **新しいコンテナを作成**：Docker がイメージを入手したあと、それを使ってコンテナを作成する。
- **ファイルシステムを割り当て、読み書き可能なレイヤをマウント**：コンテナを新しいファイルシステム上に作成し、読み込み可能な（イメージの）レイヤをイメージに追加する。
- **ネットワークとブリッジインターフェースの割り当て**：Docker コンテナがローカルホストと通信できるようにするため、ネットワーク・インターフェースを作成する。
- **IP アドレスを設定**：プールされている範囲内で利用可能な IP アドレスを探して（コンテナに）追加する。
- **指定したプロセスを実行**：アプリケーションを実行し、そして、
- **アプリケーションの出力を収集・表示**：コンテナに接続し、アプリケーションを実行したことによる標準入力・標準出力・エラーを記録・表示する。

これでコンテナが動きました！ 以降は自分でコンテナを管理し、アプリケーションと双方向でやりとりをし、利用し終えたらコンテナを停止・削除できます。

2.1.5 基礎技術

Docker は Go 言語で書かれており、これまで見てきた機能は、カーネルが持つ複数の機能を利用しています。

名前空間（namespaces）

Docker は**名前空間（ネームスペース）**と呼ばれる技術を利用し、コンテナ（container）と呼ぶワークスペース（作業空間）の分離をもたらします。Docker はコンテナごとに名前空間の集まりを作成します。

これはレイヤの分離をもたらします。つまり、コンテナを実行すると、それぞれが自身の名前空間を持ち、そこから外にはアクセスできないように見えます。

Docker が使う Linux 上の名前空間は、次の通りです。

- **pid 名前区間** ：プロセスの分離に使います（PID：プロセス ID）
- **net 名前区間** ：ネットワーク・インターフェースの管理に使います（NET：ネットワーキング）
- **ipc 名前区間** ：IPC リソースに対するアクセス管理に使います（IPC：InterProcess Communication、内部プロセスの通信）
- **mnt 名前区間** ：マウント・ポイントの管理に使います（MNT：マウント）
- **uts 名前区間** ：カーネルとバージョン認識の隔離に使います（UTS：Unix Timesharing System、Unix タイムシェアリング・システム）

コントロール・グループ(Control groups)

Linux 上の Docker は、cgroup（シーグループ）や**コントロール・グループ**と呼ばれる技術を使います。アプリケーション実行の鍵となるのは、自身が必要なリソースのみを分離します。この機能があるため、ホスト上で複数の利用者がいても、コンテナを使えます。また、コントロール・グループにより、Docker はコンテナに対して利用可能なハードウェア・リソースを共有し、必要があればコンテナが必要なリソース上限を設定できます。例えば、特定のコンテナに対する利用可能なメモリに制限を加えます。

ユニオン・ファイル・システム

ユニオン・ファイル・システム、あるいは UnionFS はファイルシステムです。これは作成したレイヤを操作しますので、非常に軽量かつ高速です。Docker はコンテナごとにブロックを構築するため、ユニオン・ファイル・システムを使います。Docker は AUFS、btrfs、vfs、DeviceMapper を含む複数のユニオン・ファイル・システム

の派生を利用できます。

コンテナの形式（フォーマット）

Docker はこれらのコンポーネントを連結し、包み込んでいます。これを**コンテナ形式（フォーマット）**と呼びます。デフォルトのコンテナ形式は libcontainer（リブコンテナ）と呼ばれています。いずれ、Docker は他のコンテナ形式、例えば BSD Jail（ビーエスディージェイル）や Solaris Zone（ソラリスゾーン）との統合をサポートするかも知れません。

2.1.6 次のステップ

Docker インストール

3 章「インストール」をご覧ください。

Docker ユーザガイド

5 章「Docker ユーザガイド」で、より深く学びましょう。

3 章

Docker Engine のインストール

Docker Engine は Linux、クラウド、Windows、OS X での動作をサポートしています。インストール方法は、以下の各ページをご覧ください。

3.1 Ubuntu

Docker は以下のオペレーティング・システムをサポートしています。

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

このページは、Docker が管理しているパッケージとインストール手順で作業します。Docker が提供する最新リリースのパッケージを使えるようにします。もし Ubuntu が管理するパッケージを使いたい場合は、Ubuntu のドキュメントをお調べください。

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ません。

3.1.1 動作条件

Docker は 64 bit でインストールされた何らかの Ubuntu バージョンを必要とします。加えて、kernel は少なくとも 3.10 以上が必要です。最新の 3.10 マイナーバージョンか、それよりも新しいバージョンを利用可能です。

3.10 よりも低いカーネルは、Docker コンテナ実行時に必要な一部の機能が足りません。古いバージョンは既知のバグがあります。その影響により、特定条件下でデータの損失や定期的なカーネルパニックを引き起こします。

現在のカーネル・バージョンを確認するには、ターミナルを開き、 uname (ユーネーム) -r を使ってカーネルのバージョンを確認します。

```
$ uname -r
3.11.0-15-generic
```

以前に Docker を apt でインストールしていた場合は、apt ソースを新しい Docker リポジトリに更新してください。

apt ソースの更新

Docker 1.7.1 以上は Docker の apt リポジトリに保管されています。apt が新しいリポジトリにあるパッケージを使えるように設定します。

1. マシンに sudo もしくは root 特権のあるユーザでログインします。

2. ターミナルのウインドウを開きます。

3. パッケージ情報を更新します。APT が https メソッドで動作することを確認し、CA 証明書がインストールされるのを確認します。

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

4. 新しい G（ジーピージー） P G 鍵を追加します。

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

5. /etc/apt/sources.list.d/docker.list ファイルを任意のエディタで開きます。

ファイルが存在しなければ、作成します。

6. 既存のエントリがあれば削除します。

7. Ubuntu オペレーティング・システム向けのエントリを追加します。

利用可能なエントリは以下の通りです。

- Ubuntu Precise 12.04 (LTS)

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

- Ubuntu Trusty 14.04 (LTS)

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

- Ubuntu Wily 15.10

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

- Ubuntu Xenial 16.04 (LTS)

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

Docker のパッケージは全てのアーキテクチャに対応していません。しかし、毎晩構築（nightly build）のバイナリは https://master.dockerproject.org/ にあります。Docker をマルチ・アーキテクチャのシステムにインストールするには、 `[arch=...]` エントリの項目を追加します。詳細は Debian Multiarch wiki[*1] をご覧ください。

8. `/etc/apt/sources.list.d/docker.list` ファイルを保存して閉じます。

9. apt パッケージのインデックスを更新します。

```
$ sudo apt-get update
```

10. 古いリポジトリが残っているのなら、パージします。

```
$ sudo apt-get purge lxc-docker
```

11. apt が正しいリポジトリから取得できるか確認します。

```
$ apt-cache policy docker-engine
```

これで `apt-get update` を実行したら、 `apt` は新しいリポジトリから取得します。

Ubuntu バージョン固有の動作条件

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

Ubuntu Trusty・Wily・Xenial では、 `linux-image-extra` カーネル・パッケージのインストールを推奨します。この `linux-image-extra` は `aufs` ストレージ・ドライバを利用可能にします。
自分のカーネル・バージョンに対応した `linux-image-extra` パッケージをインストールします。

1. Ubuntu ホスト上のターミナルを開きます。

2. パッケージ・マネージャを更新します。

```
$ sudo apt-get update
```

3. 推奨パッケージをインストールします。

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

4. Docker のインストールに進みます。

*1 https://wiki.debian.org/Multiarch/HOWTO#Setting_up_apt_sources

Ubuntu Precise 12.04 (LTS)

Ubuntu Precise では、Docker はカーネル・バージョン 3.13 が必要です。カーネルのバージョンが 3.13 よりも古い場合は、更新が必要です。環境に応じてどのパッケージが必要になるかは、次のリストをご覧ください。

- `linux-image-generic-lts-trusty` … generic の Linux カーネル・イメージ。このカーネルは AUFS が組み込み済み。Docker 実行に必要。
- `linux-headers-generic-lts-trusty` … ZFS と VirtualBox のゲスト追加に依存するようなパッケージを利用可能にします。既存のカーネルに対して headers をインストールしなければ、”trusty” カーネル向けのヘッダをスキップします。自信がなければ、安全のためにこのパッケージを導入すべきです。
- `xserver-xorg-lts-trusty , libgl1-mesa-glx-lts-trusty` … Unity/Xorg を持たない（グラフィカルではない）環境向けのオプションです。Docker をグラフィカルな環境で実行する時に 必要 です。これらのパッケージが必要な理由は、バックポートされたカーネルに関するインストール手順をご覧ください。 LTS Enablement Stack の note 5 にある各バージョンをご覧ください。

カーネルのアップグレードと追加パッケージのインストールは次のようにします。

1. Ubuntu ホスト上でターミナルを開きます。

2. パッケージ・マネージャを更新します。

```
$ sudo apt-get update
```

3. 必要なパッケージとオプションのパッケージの両方をインストールします。

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

環境に応じて、先ほどのリストにあるパッケージをインストールします。

4. ホストを再起動します。

```
$ sudo reboot
```

5. システムの再起動後、Docker のインストールに移ります。

3.1.2 インストール

インストール前に、各 Ubuntu バージョン固有の作業を終えてください。それから、以降の手順で Docker をインストールします。

1. インストールする Ubuntu に、 `sudo` 特権を持つユーザでログインします。

2. `apt` パッケージのインデックスを更新します。

```
$ sudo apt-get update
```

3. Docker をインストールします。

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

4. docker デーモンを開始します。

```
$ sudo service docker start
```

5. docker を正常にインストールしたかを確認します。

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

このコマンドは、テストイメージをダウンロードし、コンテナとして実行します。コンテナを実行時にメッセージ情報を表示して、それから終了します。

3.1.2 オプション設定

このセクションは、Ubuntu と Docker がうまく機能するようなオプション手順を紹介します。

docker グループの作成

docker デーモンは TCP ポートの代わりに Unix ソケットをバインドします。デフォルトでは、Unix ソケットは root ユーザによって所有されており、他のユーザは sudo でアクセスできます。このため、docker デーモンは常に root ユーザとして実行されています。

docker コマンド利用時に sudo を使わないようにするには、docker という名称のグループを作成し、そこにユーザを追加します。docker デーモンが起動したら、docker グループの所有者により Unix ソケットの読み書きが可能になります。

【警告】docker グループに所属するユーザは root と同等です。システム上のセキュリティに対する影響の詳細は、Docker デーモンが直面する攻撃のページをご覧ください。

docker グループを作成し、ユーザを追加するには、

1. Ubuntu に sudo 特権のあるユーザでログインします。

ログイン時のユーザ名は ubuntu ユーザかも知れません。

2. docker グループを作成し、ユーザを追加します。

```
$ sudo usermod -aG docker ubuntu
```

3. ログアウトしてから、再度ログインします。

対象ユーザが適切な権限を持つようにするためです。

4. sudo を使わずに docker の実行を確認します。

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

失敗時は、次のようなメッセージが表示されます。

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

DOCKER_HOST 環境変数をシェル上で確認します。もし設定されていれば、unset します。

メモリとスワップ利用量の調整

ユーザが Docker を実行する時、イメージ実行時に次のメッセージを表示する場合があります。

```
WARNING: Your kernel does not support cgroup swap limit. WARNING: Your
kernel does not support swap limit capabilities. Limitation discarded.
```

このメッセージを出さないようにするには、システム上でメモリとスワップの利用量（アカウンティング）を設定します。メモリとスワップ利用量の設定を有効にしますと、Docker を使っていない時、メモリのオーバーヘッドとパフォーマンスの低下を減らします。メモリのオーバーヘッドは利用可能な全メモリの 1 %程度です。パフォーマンス低下は、おおよそ 10%です。

GNU GRUB (GNU GRand Unified Bootloader) システム上で、メモリとスワップを次のように設定します。

1. Ubuntu に sudo 特権のあるユーザでログインします。

2. /etc/default/grub ファイルを編集します。

3. GRUB_CMDLINE_LINUX 値を次のように設定します。

```
GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"
```

4. ファイルを保存して閉じます。

5. GRUB を更新します。

```
$ sudo update-grub
```

6. システムを再起動します。

UFW 転送の有効化

Docker を実行するホスト上で UFW（Uncomplicated Firewall）[*1] を使っている場合、追加設定が必要になります。Docker はコンテナのネットワーク機能のためにブリッジを使用します。デフォルトでは、UFW は全ての転送（forwarding）トラフィックを破棄(drop)します。そのため、UFW が有効な状態で Docker を実行する場合、UFW の forwarding ポリシーを適切に設定しなくてはいけません。

また、UFW のデフォルト設定は incoming トラフィックを全て拒否します。他のホストからコンテナに接続したい場合、Docker のポートに対する incoming トラフィックを許可する設定をします。Docker のポートは TLS が

*1 https://help.ubuntu.com/community/UFW

有効であれば 2376 であり、そうでなければ 2375 です。デフォルトでは、TLS が有効でなければ通信は暗号化しません。Docker のデフォルトは、TLS が有効ではありません。

UFW を設定するには、Docker ポートに対する incoming 接続を許可します。

1. Ubuntu に sudo 特権のあるユーザでログインします。

2. UFW のインストールと有効化を確認します。

```
$ sudo ufw status
```

3. /etc/default/ufw を開き、編集します。

```
$ sudo nano /etc/default/ufw
```

4. DEFAULT_FOWRARD_POLICY ポリシーを設定します。

```
DEFAULT_FORWARD_POLICY="ACCEPT"
```

5. ファイルを保存して閉じます。

6. UFW を新しい設定を使って再読み込みします。

```
$ sudo ufw reload
```

7. Docker ポートの incoming 接続を許可します。

```
$ sudo ufw allow 2375/tcp
```

Docker が使う DNS サーバの設定

Ubuntu や Ubuntu 派生システムのデスクトップを動かすシステムは、デフォルトで/etc/resolv.conf ファイルで使用する nameserver は 127.0.0.1 です。NetworkManager も dnsmasq をセットアップする時は、/etc/resolv.conf を nameserver 127.0.0.1 に設定します。

デスクトップ・マシンでコンテナを起動時、このような設定であれば、次の警告が出ます。

```
WARNING: Local (127.0.0.1) DNS resolver found in resolv.conf and containers
can't use it. Using default external servers : [8.8.8.8 8.8.4.4]
```

この警告は、Docker コンテナがローカルの DNS サーバを使えないためです。そのかわり Docker はデフォルトで外部のネームサーバを使います。

警告を出ないようにするには、Docker コンテナが使うための DNS サーバを指定します。あるいは NetworkManager で dnsmasq を無効にもできます。dnsmasq を無効にしたら、同一ネットワークの DNS 名前解決が遅くなるかも知れません。

Docker が使う DNS サーバの指定方法は、次の通りです。

1. Ubuntu に sudo 特権のあるユーザでログインします。

2. /etc/default/docker ファイルを開き、編集します。

```
$ sudo nano /etc/default/docker
```

4. Docker の設定を追加します。

```
DOCKER_OPTS="--dns 8.8.8.8"
```

8.8.8.8 を 192.168.1.1 のようなローカルの DNS サーバに置き換えます。複数の DNS サーバも指定できます。その場合は、次の例のようにスペースで分離します。

```
--dns 8.8.8.8 --dns 192.168.1.1
```

【警告】この作業を PC 上で行う場合は様々なネットワークに接続するため、パブリック DNS サーバを選択してください。

4. ファイルを保存して閉じます。

5. Docker デーモンを再起動します。

```
$ sudo restart docker
```

あるいは、先ほどの手順とは別の方法として、NetworkManager（ネットワークマネージャ）で dnsmasq（DNS マスカレード）を無効化する方法もあります（ネットワークが遅くなるかも知れません）。

1. /etc/NetworkManager/NetworkManager.conf ファイルを開き、編集します。

```
$ sudo nano /etc/NetworkManager/NetworkManager.conf
```

2. dns=dnsmasq 行をコメントアウトします。

```
dns=dnsmasq
```

3. ファイルを保存して閉じます。

4. NetworkManager と Docker の両方を再起動します。

```
$ sudo restart network-manager
$ sudo restart docker
```

ブート時の Docker 起動設定

Ubuntu 15.04 以上はサービス・マネージャに systemd（システムディー）を使って起動します。14.10 以下のバージョンでは upstart（アップスタート）です。

15.04 以上で docker デーモンをブート時に起動するようにするには、次のように実行します。

```
$ sudo systemctl enable docker
```

14.10 以下では、自動的に upstart を使って Docker デーモンをブート時に起動する設定がインストール時に行われます。

3.1.3 Docker のアップグレード

Docker の最新版をインストールするには、apt-get を使います。

```
$ sudo apt-get upgrade docker-engine
```

3.1.4 アンインストール

Docker パッケージをアンインストールします。

```
$ sudo apt-get purge docker-engine
```

Docker パッケージと必要のない依存関係をアンインストールします。

```
$ sudo apt-get autoremove --purge docker-engine
```

上記のコマンドは、イメージ、コンテナ、ボリュームやホスト上の設定ファイルを削除しません。イメージ、コンテナ、ボリュームを削除するには次のコマンドを実行します。

```
$ rm -rf /var/lib/docker
```

ユーザが作成した設定ファイルは、手動で削除する必要があります。

4 章

Docker Engine クイックスタート

このクイックスタートを進めるにあたり、Docker Engine のインストール完了を前提にしています。Docker Engine がインストール済み・設定済みかを確認するには、次のコマンドを実行します。

```
# インストールした Docker の正常動作を確認
$ docker info
```

インストールに成功しているのであれば、システム情報を表示します。もしも `docker: command not found`（訳：docker コマンドが見つかりません）や `/var/lib/docker/repositories: permission denied`（訳：権限がありません）のような表示が出る場合は、Docker のインストールが不完全か、コマンドでマシン上の Docker Engine にアクセスする権限がありません。Docker Engine の標準インストールでは、docker コマンドを実行するには docker グループに所属するユーザ、もしくは root の必要があります。

Docker Engine のシステム設定によっては、各 `docker` コマンドの前に `sudo` が必要になる場合があります `docker` コマンドで `sudo` を使わないようにする方法の 1 つに、`docker` という名称の Unix グループを作成し、ユーザを `docker` グループに追加して `docker` コマンドを使えるようにできます。

Docker Engine のインストールや `sudo` 設定に関しては、 2章「インストール」を参照ください。

4.1 クイックスタート・ガイド

4.1.1 構築済みイメージのダウンロード

ubuntu イメージをダウンロード（pull）するには、次のように実行します。

```
# ubuntu イメージのダウンロード（pull）
$ docker pull ubuntu
```

このコマンドは Docker Hub 上の ubuntu イメージをローカルのイメージ・キャッシュにダウンロードします。イメージを検索するには `docker search`（サーチ） コマンドを実行します。詳しい情報はイメージの検索のページをご覧ください。

イメージのダウンロードに成功したら、12 文字のハッシュ `539c0211cd76: Download complete`（ダウンロード コンプリート） が表示されます。これは**イメージ ID** を短くしたものです。この短いイメージ ID（short image ID）（ショート イメージアイディー）は、**完全イメージ ID**（full image ID）（フル アイディー）の先頭から 12 文字です。完全イメージ ID を確認するには `docker inspect`（インスペクト） か `docker images --no-trunc=true`（ノートランケート トゥルー） を実行します。

ダウンロードしたイメージの一覧を表示するには docker images を実行します。

4.1.2 対話型シェルの実行

Ubuntu イメージの対話型シェルを使うには、次のように実行します：

```
$ docker run -i -t ubuntu /bin/bash
```

-i フラグは対話型（interactive；インタラクティブ） のコンテナを起動します。-t フラグは疑似ターミナル（pseudo-TTY） を起動し、ｓｔｄｉｎ と ｓｔｄｏｕｔ（標準入出力）を接続（attach）します。イメージ名は ubuntu です。コマンド /bin/bash を使ってログインできます。

シェルを終了せずに tty を取り外す（detach）には、エスケープ・シーケンス[*1] Ctrl-p + Ctrl-q を使います。コンテナから出たあとも、停止するまでコンテナは存在し続けます。

4.1.3 Docker を他のホスト・ポートや Unix ソケットに接続

【警告】docker デーモンが標準で利用する TCP ポートと Unix docker ユーザ・グループの変更は、ホスト上の非 root ユーザが root アクセスを得られるという、セキュリティ・リスクを増やします。docker に対する管理を確実に行ってください。TCP ポートの利用時、ポートにアクセスできる誰でも Docker に対する完全なアクセスを可能です。そのため、オープンなネットワーク上での利用は望ましくありません。

Docker デーモンに-H オプション使用したら、指定した IP アドレスとポートをリッスンします（ポートを開きます)。標準では、unix:///var/run/docker.sock[*2] をリッスンし、ローカルの root ユーザのみ接続できます。これを 0.0.0.0:2375 や特定のホスト IP を指定することで、誰でもアクセス可能にできますが、**推奨されていません**。理由は、デーモンが稼働しているホスト上の root アクセスを、誰もが簡単に得られるためです。

同様に、Docker クライアントも-H オプションを使い、任意のポートに接続可能です。Docker クライアントは、Linux 版では unix:///var/run/docker.sock に接続し、Windows 版では tcp://127.0.0.1:2376 に接続します。

-H は次の書式でホストとポートを割り当てます：

```
tcp://[ホスト]:[ポート番号][パス] または unix:///パス
```

例：

- tcp:// → 127.0.0.1 に TCP 接続時、TLS 暗号化が有効であればポート 2376 を、通信がプレーンテキストの場合（暗号化していない）はポート 2375 を使います。
- tcp://host:2375 → 対象ホストのポート 2375 に TCP で接続します。
- tcp://host:2375/パス → 対象ホストのポート 2375 に TCP で接続し、あらかじめリクエストのパスを追加します。
- unix://ソケット/の/パス → ソケットのパスにある Unix ソケットに接続します。

---

*1 この場合、キーボードから入力する特殊な文字で制御すること。

*2 覚え方は「unix://」のプロトコルを指定した後ろに「/var/run/docker.sock」を指定しています。「tcp://」のようにプロトコルを指定するのと同じです。

-H の後ろに何も指定しなければ、標準では-H を指定していないのと同じ挙動になります。
また、-H は TCP の指定を省略できます。

`host:` または `host:port` または `:port`

Docker をデーモン・モードで起動するには、次のようにします。

```
$ sudo <path to>/docker  daemon -H 0.0.0.0:5555 &
```
（docker バイナリのパス）

ubuntu イメージをダウンロードするには、次のようにします。

```
$ docker -H :5555 pull ubuntu
```

また、複数の-H オプションを使えます。例えば TCP と Unix ソケットの両方をリッスンしたい場合に使えます。

```
# docker をデーモン・モードで実行
$ sudo <path to>/docker daemon -H tcp://127.0.0.1:2375 -H unix:///var/run/docker.sock &
# 標準の Unix ソケットを使い、Ubuntu イメージをダウンロード
$ docker pull ubuntu
# あるいは、TCP ポートを使用
$ docker -H tcp://127.0.0.1:2375 pull ubuntu
```

4.1.4 長時間動作するワーカー・プロセスの開始

```
# とても便利な長時間動作プロセスの開始
$ JOB=$(docker run -d ubuntu /bin/sh -c "while true; do echo Hello world; sleep 1; done")

# これまでのジョブの出力を収集
$ docker logs $JOB
```
（ログズ）

```
# ジョブの停止(kill)
$ docker kill $JOB
```
（キル）

4.1.5 コンテナの一覧

```
$ docker ps # 実行中のコンテナのみリスト表示
$ docker ps -a # 全てのコンテナをリスト表示
```

4.1.6 コンテナの制御

```
# 新しいコンテナの起動
$ JOB=$(docker run -d ubuntu /bin/sh -c "while true; do echo Hello world; sleep 1; done")

# コンテナの停止
$ docker stop $JOB

# コンテナの起動
```

```
$ docker start $JOB

# コンテナの再起動
$ docker restart $JOB

# コンテナを SIGKILL で停止
$ docker kill $JOB

# コンテナを削除
$ docker stop $JOB # Container must be stopped to remove it
$ docker rm $JOB
```

4.1.7 TCP ポートにサービスを割り当て

```
# コンテナにポート 4444 を割り当て、netcat でリッスンする
$ JOB=$(docker run -d -p 4444 ubuntu:12.10 /bin/nc -l 4444)

# どの外部ポートがコンテナに NAT されているか？
$ PORT=$(docker port $JOB 4444 | awk -F: '{ print $2 }')

# 公開ポートに接続
$ echo hello world | nc 127.0.0.1 $PORT

# ネットワーク接続の動作を確認
$ echo "Daemon received: $(docker logs $JOB)"
```

4.1.8 コンテナの状態を保存

現在のコンテナの状態をイメージとして保存（commit）するには、 docker commit コマンドを使います。

```
$ docker commit <コンテナ> <何かの名前>
```

コンテナのコミットとは、元になったイメージと現在のコンテナの差分情報のみを、Docker Engine が保存します。どのようなイメージがあるかを確認するには、次のコマンドを実行します。

```
# イメージ一覧を表示
$ docker images
```

コミットによって新しいイメージを手に入れました。これを使い、新しいインスタンス（訳者注：コンテナのこと）を作成可能です。

5 章

Docker Engine ユーザガイド

5.1 はじめに

5.1.1 ユーザガイドについて

当ガイドは皆さんの環境に Docker Engine を導入するための基礎となります。Docker Engine に関する以下の使い方を学びます。

- アプリケーションの Docker 化（Dockerize（ドッカーライズ））
- 自分自身でコンテナを実行
- Docker イメージの構築
- Docker イメージを他人と共有
- 他にも多くのことを！

Docker Engine の基本と Docker がサポートするプロダクトを理解できるようにするため、このガイドは主なセクションを分けています。

アプリケーションの Docker 化：”Hello world”

「アプリケーションをコンテナの中で実行するには？」

Docker Engine は、アプリケーションを強力にするコンテナ化プラットフォームです。アプリケーションを Docker に対応する方法と実行の仕方を学びます。

コンテナの操作

「コンテナを管理するには？」

アプリケーションを Docker コンテナで実行できるようになったら、これらのコンテナの管理方法を学びます。コンテナの調査、監視、管理の仕方を理解します。

Docker イメージの操作

「自分のイメージにアクセス、共有、構築するには？」

Docker の使い方を学んだら、次のステップに進みます。Docker で自分のアプリケーション・イメージを構築する方法を学びます。

コンテナのネットワーク

これまで Docker コンテナの中に個々のアプリケーションを構築する方法を理解しました。次は Docker ネットワークでアプリケーション・スタックを構築する方法を学びます。

コンテナ内のデータ管理

Docker コンテナ間を接続する方法を学んだら、次はコンテナの中にあるデータ、ボリューム、マウントに関する管理方法を学びます。

5.1.2 Engine を補う Docker プロダクト

多くの場合、ある強力な技術は更に技術を生み出します。何かをより簡単に入手できるように、より簡単に使えるように、より強力にするように、等です。生み出されたものは共有されるという特徴があります。つまり、結果として中心にある技術を補強するのです。以下の Docker プロダクトは、中心となる Docker Engine の機能を拡張します。

Docker Hub

Docker Hub（ハブ）は Docker の中心となる場所（ハブ）です。公開用の Docker イメージを提供し、Docker 環境の構築と管理の手助けとなるサービスを提供します。

Docker Machine

Docker Machine（マシーン）は Docker Engine を起動し、迅速に実行する手助けをします。Machine で Docker Engine をセットアップできるのは、自分のコンピュータ上や、クラウド事業者上だけではありません。データセンタでもセットアップできます。セットアップ後は Docker クライアントが安全に通信できるように設定します。

Docker Compose

Docker Compose（コンポーズ）はアプリケーションの構成を定義します。コンテナと設定、リンク、ボリュームに関する情報を、1 つのファイル上で記述します。コマンド 1 つ実行するだけで、全てのをセットアップし、アプリケーションを実行します。

Docker Swarm

Docker Swarm（スウォーム）は複数の Docker Engine をまとめて、1 つの仮想的な Docker Engine のように振る舞います。標準 Docker API に対応しているため、Docker で利用可能なツールであれば、複数のホスト上に透過的なスケールアップが可能です。

5.2 使用例を学ぶ

5.2.1 コンテナで Hello world（ハローワールド）

Docker とは一体何なのでしょうか。Docker はコンテナ内に作成した世界で、アプリケーションを実行可能にします。コンテナ内でアプリケーションを実行するには、docker run（ラン） コマンドを実行するだけです。

Docker システムの設定によっては、ガイドにおける各ページの docker コマンドで sudo が必要になる場合があります。この挙動を回避するには、システム管理者に対して docker という名称の Unix グループを作成し、そこにユーザを追加するようご依頼ください。

Hello world の実行

まず hello world コンテナを実行しましょう。

```
$ docker run ubuntu:14.04 /bin/echo 'Hello world'
Hello world
```

初めてコンテナを実行しました！　この例では、以下の作業を行いました。

- docker run でコンテナを実行します（run は「実行」の意味)。
- ubuntu は実行するイメージです。この例では Ubuntu オペレーティング・システムのイメージです。イメージを指定したら、Docker はまずホスト上にイメージがあるかどうか確認します。イメージがローカルに存在していなければ、パブリック・イメージ・レジストリである Docker Hub からイメージを取得（pull）します。
- /bin/echo は新しいコンテナ内で実行するコマンドです。

コンテナを起動するとは、Docker が新しい Ubuntu 環境を作成し、その中で /bin/echo コマンドを実行し、その結果を出力します。

```
Hello world
```

それでは、表示後のコンテナはどのような状況でしょうか。Docker コンテナが実行されていたのは、指定したコマンドを処理していた間のみです。この例では、コマンドを実行したのち、直ちにコンテナが停止しました。

インタラクティブなコンテナを実行

新しいコマンドを指定して、別のコンテナを起動しましょう。

```
$ docker run -t -i ubuntu:14.04 /bin/bash
root@af8bae53bdd3:/#
```

この例は、次の処理を行います。

- docker run コマンドでコンテナを実行します。
- ubuntu イメージを使って起動します。
- -t フラグは新しいコンテナ内に疑似ターミナル（pseudo-tty（シュードティーティーワイ））を割り当てます。

- -i フラグはコンテナの標準入力 (STDIN)を取得し、双方向に接続できるようにします。[*1]
- /bin/bash はコンテナ内で Bash シェルを起動します。

コンテナを起動したら、次のようなコマンド・プロンプトが表示されます。

```
root@af8bae53bdd3:/# pwd
/
root@af8bae53bdd3:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
```

この例は：

- pwd[*2] を実行し、現在のディレクトリが表示されます。ここでは / ルート・ディレクトリにいることが分かります。
- ls[*3] はルートディレクトリ以下のディレクトリ一覧を表示します。典型的な Linux ファイル・システムのように見えます。

これで、コンテナ内で遊べます。終わったら exit コマンドまたは Ctrl-D を入力して終了できます。

```
root@af8bae53bdd3:/# exit
```

先ほど作成したコンテナと同様に、Bash シェルのプロセスが終了したら、コンテナは停止します。

Docker 化した Hello world の起動

デーモンとして実行するコンテナを作成しましょう。

```
$ docker run -d ubuntu:14.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"
1e5535038e285177d5214659a068137486f96ee5c2e85a4ac52dc83f2ebe4147
```

この例では：

- docker run はコンテナを実行します。
- -d フラグはバックグラウンドで（デーモン化して）コンテナを実行します。
- ubuntu は実行しようとしているイメージです。

最後に、実行するコマンドを指定します：

```
/bin/sh -c "while true; do echo hello world; sleep 1; done"
```

*1 正確には標準エラー(STDOUT)も含めた標準入出力を扱います。

*2 Print Working Directory = 作業ディレクトリの表示という意味です。

*3 list segment = セグメント一覧の意味です。

出力は先ほどのように hello world を表示せず、文字列を表示します。

```
1e5535038e285177d5214659a068137486f96ee5c2e85a4ac52dc83f2ebe4147
```

コンテナ ID は長くて扱いにくいものです。あとで短い ID を扱います。こちらを使えば、コンテナをより簡単に操作できます。

この**コンテナ ID** を使い、hello world デーモンで何が起こっているのかを調べます。

はじめに、コンテナが実行中であることを確認しましょう。docker ps コマンドを実行します。docker ps コマンドは、Docker デーモンに対し、デーモンが知っている全てのコンテナ情報を問い合わせます。

```
$ docker ps
CONTAINER ID  IMAGE         COMMAND               CREATED        STATUS       PORTS NAMES
1e5535038e28  ubuntu:14.04  /bin/sh -c 'while tr  2 minutes ago  Up 1 minute        insane_babbage
```

この例はデーモン化したコンテナを見ています。 docker ps は便利な情報を返します。

- 1e5535038e28 はコンテナ ID の短いバージョンです。
- ubuntu は使用したイメージです。
- コマンド、状態、コンテナに自動で割り当てられた名前は insane_babbage[*1] です。

Docker はコンテナ開始する時、自動的に名前を作成します。自分自身で名前を指定する方法は、後ほど紹介します。

これでコンテナが実行中だと分かりました。しかし、実行時に指定した処理が正しく行われているでしょうか。コンテナの中でどのような処理が行われているか確認するには、docker logs（ログズ） を使います。

コンテナ名 insane_babbage を指定しましょう。

```
$ docker logs insane_babbage
hello world
hello world
hello world
. . .
```

この例では：

- docker logs でコンテナ内をのぞき込んだら、 hello world を返します。

素晴らしいです！ デーモンとして動いています。初めて Docker 化（Dockerized）したアプリケーションを作成しました！

次は docker stop（ストップ） コマンドでデタッチド・コンテナ（バックグラウンドで動作しているコンテナ）を停止します。

*1 コンテナ名は形容詞＋歴史上の有名な科学者・ハッカーの組みあわせで自動生成します。詳細：https://github.com/docker/docker/blob/master/pkg/namesgenerator/names-generator.go

```
$ docker stop insane_babbage
insane_babbage
```

docker stop コマンドは、Docker に対して丁寧にコンテナを停止するよう命令します。処理が成功したら、停止したコンテナ名を表示します。
docker ps コマンドを実行して、動作確認しましょう。

```
$ docker ps
CONTAINER ID  IMAGE         COMMAND               CREATED        STATUS       PORTS NAMES
```

素晴らしいですね。コンテナが停止しました。

次のステップ

ここまでは docker run コマンドを使い、初めてのコンテナを起動しました。フォアグラウンドで動作する、双方向に操作可能なコンテナを実行しました。また、バックグラウンドで動作するデタッチド・コンテナも実行しました。この過程で、複数の Docker コマンドを学びました。

docker ps - コンテナの一覧を表示。
docker logs - コンテナの標準出力を表示。
docker stop - 実行中のコンテナを停止。

以上で、Docker の基本と高度な処理を学びました。次はシンプルなアプリケーションの実行を通し、Docker クライアントを使って実際のウェブアプリケーションを構築しましょう。

5.2.2 シンプルなアプリケーションの実行

前節では、docker run コマンドを使い、初めてのコンテナを起動しました。そして、フォアグラウンドで**インタラクティブなコンテナ**（interactive container）を実行しました。また、バックグラウンドで**デタッチド・コンテナ**（detached container）を実行しました。この過程で複数の Docker コマンドを学びました。

docker ps - コンテナの一覧を表示。
docker logs - コンテナの標準出力を表示。
docker stop - 実行中のコンテナを停止。

Docker クライアントについて学ぶ

気が付いていないかも知れませんが、Bash ターミナル上で毎回 docker と入力し、既に Docker クライアントを利用していました。クライアントとはシンプルなコマンドライン・クライアントであり、コマンドライン・インターフェース（CLI）とも呼びます。クライアントを使った各種の動作にはコマンド（命令）を使います。各コマンドには一連のフラグや引数を持ちます。

```
# 使い方: [sudo] docker [サブコマンド] [フラグ] [引数] ..
# 例:
$ docker run -i -t ubuntu /bin/bash
```

実際に動作するかどうかは docker version（バージョン）コマンドを使います。現在インストールしている Docker クライアントとデーモンのバージョン情報を確認できます。

```
$ docker version
```

このコマンドは使用している Docker クライアントとデーモンのバージョンを表示するだけではありません。Go 言語（Docker を動かすプログラミング言語）のバージョンも表示します。

```
Client:
 Version:      1.8.1
 API version:  1.20
 Go version:   go1.4.2
 Git commit:   d12ea79
 Built:        Thu Aug 13 02:35:49 UTC 2015
 OS/Arch:      linux/amd64

Server:
 Version:      1.8.1
 API version:  1.20
 Go version:   go1.4.2
 Git commit:   d12ea79
 Built:        Thu Aug 13 02:35:49 UTC 2015
 OS/Arch:      linux/amd64
```

Docker コマンドの使い方を表示

特定の Docker コマンドに対する使い方も表示できます。help（ヘルプ） は使い方の詳細を表示します。利用可能なコマンドの一覧を表示するには、次のように実行します：

```
$ docker --help
```

一般的な使い方は、コマンドラインで`--help` フラグを指定します。

```
$ docker attach --help

Usage: docker attach [OPTIONS] CONTAINER

Attach to a running container

  --help=false        Print usage
  --no-stdin=false    Do not attach stdin
  --sig-proxy=true    Proxy all received signals to the process
```

各コマンドの更に詳細や例については、コマンド・リファレンスをご覧ください。

Docker でウェブ・アプリケーションを実行

ここまでは docker クライアントについて少しだけ学びました。次は多くのコンテナの実行という、より重要なことを学びます。これまで実行したコンテナのほとんどは、いずれも何かに役に立つ処理を行いませんでした。今度は、Docker を使ったウェブ・アプリケーションの実行に移ります。

ウェブ・アプリケーションとして、Python（パイソン） の Flask（フラスコ） アプリケーションを実行します。`docker run` コマンドで開始します。

```
$ docker run -d -P training/webapp python app.py
```

コマンドの実行内容を精査します。-d と-P という２つのフラグを指定しました。-d フラグは既出であり、コンテナをバックグラウンドで実行するよう Docker に命令します。-P は新しいフラグで、コンテナ内部のネットワークで必要なポートを、ホスト側にマップする（割り当てる）よう Docker に命令します。これにより、ウェブ・アプリケーションを参照できます。

ここではイメージ raining/webapp を指定しました。このイメージは事前に構築しておいたイメージであり、シンプルな Python Flask ウェブ・アプリケーションが入っています。

最後にコンテナに対して python app.py を実行するコマンドを指定しました。これでウェブ・アプリケーションが起動します。

docker run コマンドについて、より詳細を知りたい場合はコマンド・リファレンスと Docker Run リファレンス をご覧ください。

ウェブ・アプリケーションのコンテナを表示

さて、docker ps コマンドを使い、実行中のコンテナを表示できます。

```
$ docker ps -l
CONTAINER ID  IMAGE                    COMMAND       CREATED        STATUS        PORTS
   NAMES
bc533791f3f5  training/webapp:latest  python app.py 5 seconds ago  Up 2 seconds  0.0.0.0:49155->5000/tcp
 nostalgic_morse
```

docker ps コマンドに新しいフラグ-l を指定しています。これは、最後に開始したコンテナの詳細を返すよう、docker ps コマンドに命令します。

標準では、docker ps コマンドは実行中のコンテナ情報のみ表示します。停止したコンテナの情報も表示したい場合は、-a[*1] フラグを使います。

前節で見てきた詳細に加え、PORTS 列に重要な情報が追加されています。

```
PORTS
0.0.0.0:49155->5000/tcp
```

docker run コマンドに-P フラグを渡したら、Docker はイメージからホスト側に対して、必要なポートを**露出**(エクスポーズ)**(expose)** します。

Docker イメージで露出ポートを指定する方法は、 イメージの構築方法の節で学びます。

今回の場合、Docker はコンテナのポート 5000（Python Flask の標準ポート）をホスト上のポート 49115 上に公開しました。

*1 -a は all（全て）の意味です。

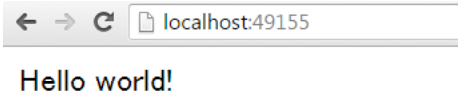
Docker は、ネットワーク・ポートの割り当て設定を変更可能です。今回の例では、-P フラグは -p 5000 を指定するショートカットにあたります。これは、コンテナの中のポート 5000 を、ローカルの Docker ホスト上のハイポート（典型的な 32768 〜 61000 の範囲にある一時利用ポート[*1]）に割り当てます。あるいは、 -p フラグを使うことで、Docker コンテナに割り当てるポートの指定も可能です。例：

```
$ docker run -d -p 80:5000 training/webapp python app.py
```

これはローカルホスト上のポート 80 を、コンテナ内のポート 5000 に割り当てます。もしかすると、次の疑問を持つでしょう。Docker コンテナをハイポートにマッピングするのではなく、常に 1:1 のポート割り当てを使わないのかと。ですが、ローカルホスト上の各ポートに 1:1 で割り当て可能なポートは 1 つだけだからです。

例えば、2 つの Python アプリケーションを実行したいとします。いずれもコンテナの中でポート 5000 を使うものです。この場合 Docker のホスト上で、ポート 5000 にアクセスできるコンテナは常に 1 つだけです。

それではウェブ・ブラウザからポート 49155 を表示してみます。



Hello world!

Python アプリケーションが動いています！

Mac OS X や Windows または Linux 上の仮想マシンを使っている場合は、ローカルホスト上で仮想マシンが使っている IP アドレスを確認する必要があります。コマンドラインや端末アプリケーションを使い docker-machine ip <仮想マシン名> を実行します。例：

```
$ docker-machine ip my-docker-vm
192.168.99.100
```

この例では、ブラウザで http://192.168.99.100:49155 を開きます。

network port でショートカット

割り当てたポートを確認するのに docker ps コマンドを使うのは、少々面倒です。そこで、Docker の docker port という便利なソートカットを使いましょう。docker port でコンテナ ID や名前を指定したら、公開ポートに割り当てられているポート番号が分かります。

```
$ docker port nostalgic_morse 5000
0.0.0.0:49155
```

この例では、コンテナ内のポート 5000 が、外部の何番ポートに割り当てられたか分かります。

ウェブ・アプリケーションのログ表示

アプリケーションで何が起こっているのか、より詳しく見てみましょう。これまで学んだ docker logs コマンドを使います。

*1 具体的なポートの範囲は Linux カーネルのパラメータ指定に依存します。

```
$ docker logs -f nostalgic_morse
* Running on http://0.0.0.0:5000/
10.0.2.2 - - [23/May/2014 20:16:31] "GET / HTTP/1.1" 200 -
10.0.2.2 - - [23/May/2014 20:16:31] "GET /favicon.ico HTTP/1.1" 404 -
```

今回は新しい -f フラグを使いました。これは docker logs コマンドに対して tail -f コマンドのように動作するもので、コンテナの標準出力を見ます。ここではポート 5000 で動作している Flask アプリケーションに対する接続ログを表示します。

アプリケーション・コンテナのプロセスを表示

コンテナのログに加え、docker top（トップ）コマンドを使えば、内部で実行しているプロセスを確認できます。

```
$ docker top nostalgic_morse
PID                 USER                COMMAND
854                 root                python app.py
```

ここでは python app.py コマンドだけが、コンテナ内のプロセスとして動作していることが分かります。

ウェブ・アプリケーション・コンテナの調査

最後に、Docker コンテナに低レベルでアクセスするには、docker inspect（インスペクト）コマンドを使います。指定したコンテナに対する便利な構成情報やステータス情報を、JSON（ジェイソン）形式で得られます。

```
$ docker inspect nostalgic_morse
```

実行後、次のような JSON 出力例を表示します。

```
[{
    "ID": "bc533791f3f500b280a9626688bc79e342e3ea0d528efe3a86a51ecb28ea20",
    "Created": "2014-05-26T05:52:40.808952951Z",
    "Path": "python",
    "Args": [
       "app.py"
    ],
    "Config": {
       "Hostname": "bc533791f3f5",
       "Domainname": "",
       "User": "",
...
```

あるいは、必要となる特定の情報のみ表示するように、情報を絞り込めます。次の例では、コンテナの IP アドレスのみ表示します。

```
$ docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' nostalgic_morse
172.17.0.5
```

ウェブ・アプリケーション・コンテナの停止

ここまではウェブ・アプリケーションが動作するのを確認しました。次は docker stop（ストップ）コマンドを使い、nostalgic_morse という名前のコンテナを指定します。

```
$ docker stop nostalgic_morse
nostalgic_morse
```

docker ps コマンドを使い、コンテナの停止を確認します。

```
$ docker ps -l
```

ウェブ・アプリケーション・コンテナの再起動

おっと！ コンテナを停止後に、他の開発者がコンテナを元に戻して欲しいと言ってきました。ここでは2つの選択肢があります。新しいコンテナを起動するか、あるいは古いものを再起動するかです。先ほどのコンテナを元に戻してみましょう。

```
$ docker start nostalgic_morse
nostalgic_morse
```

ここで素早く docker ps -l を再度実行したら、実行していたコンテナが復帰し、コンテナの URL をブラウザで開けば、アプリケーションが応答します。

docker restart コマンドも利用可能です。こちらはコンテナの停止と起動を行います。

ウェブ・アプリケーション・コンテナの削除

同僚は作業を終え、コンテナがもう不要との連絡がありました。これで、docker rm コマンドで削除できます。

```
$ docker rm nostalgic_morse
Error: Impossible to remove a running container, please stop it first or use -f
2014/05/24 08:12:56 Error: failed to remove one or more containers
```

何が起こったのでしょうか？ 実行中かもしれないコンテナを間違って削除しないように、保護されているからです。先にコンテナを停止してから、再び実行します。

```
$ docker stop nostalgic_morse
nostalgic_morse
$ docker rm nostalgic_morse
nostalgic_morse
```

今度はコンテナを停止し、削除しました。

常に、最後にコンテナを削除するのを忘れないでください。[*1]

*1 訳者注：コンテナ用の読み書き可能なレイヤが、誰からも必要とされないまま残り続け、ディスク容量を消費するのを避けるためです。

次のステップ

ここまでは Docker Hub からダウンロードしたイメージのみを使ってきました。次は、自分でイメージを構築し、共有する方法を紹介します。

5.2.3 イメージの構築

Docker イメージはコンテナの土台（基盤）です。docker run を実行する度に、どのイメージを使うか指定します。ガイドの前セクションでは、既存の ubuntu イメージと training/webapp イメージを使いました。

Docker はダウンロードしたイメージを Docker ホスト上[*1] に保管しており、それらを見ることができます。もしホスト上にイメージがなければ、**Docker はレジストリ**[*2] からイメージをダウンロードします。標準のレジストリは **Docker Hub（ハブ） レジストリ**（https://hub.docker.com/）です。

このセクションでは Docker イメージについて、次の内容を含めて深掘りします。

- ローカルの Docker ホスト上にあるイメージの管理と操作
- 基本イメージの作成
- イメージを Docker Hub レジストリにアップロード

ホスト上のイメージ一覧を表示

ローカルのホスト上にあるイメージの一覧を表示しましょう。表示するには docker images（イメージズ） コマンドを使います。

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              14.04               1d073211c498        3 days ago          187.9 MB
busybox             latest              2c5ac3f849df        5 days ago          1.113 MB
training/webapp     latest              54bb4e8718e8        5 months ago        348.7 MB
```

これまでのガイドで使用したイメージを表示します。それぞれ、コンテナでイメージを起動する時に Docker Hub からダウンロードしたものです。イメージの一覧表示には、３つの重要な情報が表示されます。

- どの**リポジトリ**から取得したのか（例：ubuntu）
- 各イメージの**タグ**（例：14.04）
- イメージごとの**イメージ ID**

サード・パーティ製の dockviz tool[*3] や image layers[*4] サイトでイメージ・データを可視化できます。

リポジトリによっては複数の派生イメージを持つ場合があります。先ほどの ubuntu イメージの場合は、Ubuntu 10.04、12.04、12.10、13.03、13.10 という、複数の異なる派生イメージがあります。それぞれの違いを**タグ (tag)** によって識別します。そして、次のようにイメージに対するタグとして参照できます。

---

*1 訳者注：Docker Engine デーモンが動くホストです。

*2 訳者注：イメージの保管庫という意味です。

*3 https://github.com/justone/dockviz

*4 https://imagelayers.io/

```
ubuntu:14.04
```

そのため、コンテナを実行する時は、次のようにタグ付けされたイメージを参照できます。

```
$ docker run -t -i ubuntu:14.04 /bin/bash
```

あるいは Ubuntu 14.04 イメージを使いたい場合は、次のようにします。

```
$ docker run -t -i ubuntu:12.04 /bin/bash
```

タグを指定しない場合、ここでは ubuntu しか指定しなければ、Docker は標準で ubuntu:latest イメージを使用します。

常に ubuntu:14.04 のようにイメージに対するタグを指定すべきです。タグの指定こそが、確実にイメージを使えるようにする手法だからです。トラブルシュートやデバッグに便利です。

新しいイメージの取得

それでは、新しいイメージをどうやって取得できるのでしょうか。Docker は Docker ホスト上に存在しないイメージを使う時、自動的にイメージをダウンロードします。しかしながら、コンテナを起動するまで少々時間がかかる場合があります。イメージをあらかじめダウンロードしたい場合は、docker pull（プル）コマンドを使えます。以下は centos（セントオーエス）イメージをダウンロードする例です。

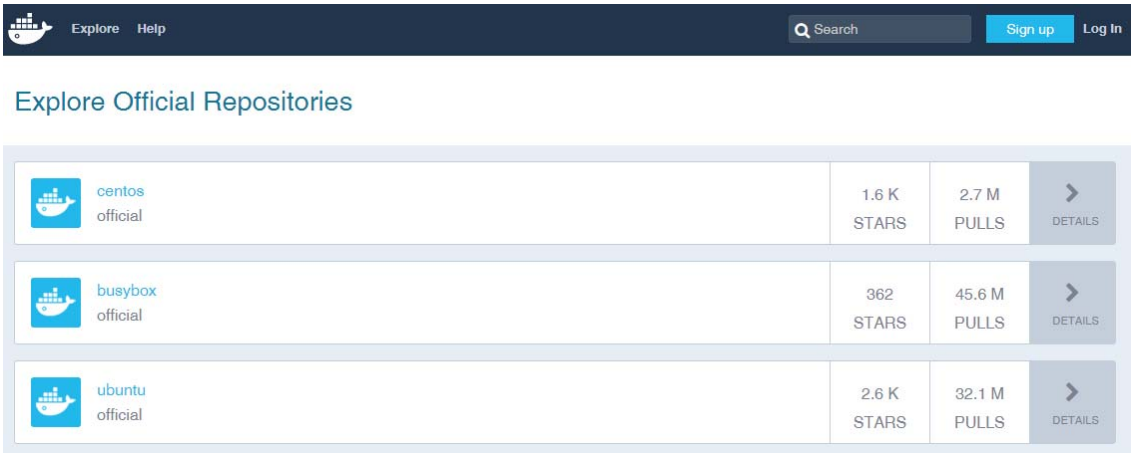
```
$ docker pull centos
Pulling repository centos
b7de3133ff98: Pulling dependent layers
5cc9e91966f7: Pulling fs layer
511136ea3c5a: Download complete
ef52fb1fe610: Download complete
...

Status: Downloaded newer image for centos
```

イメージの各レイヤを取得しているのが見えます。コンテナを起動する時、このイメージを使えばイメージのダウンロードのために待つ必要はありません。

```
$ docker run -t -i centos /bin/bash
bash-4.1#
```

イメージの検索

Docker の特長の 1 つに、様々な目的の Docker イメージが多くの方によって作られています。大部分が Docker Hub にアップロードされています。これらイメージは Docker Hub ウェブサイト（https://hub.docker.com/explore/）から検索できます。



イメージを検索するには、コマンドライン上で docker search（サーチ）コマンドを使う方法もあります。チームでウェブ・アプリケーションの開発のために Ruby と Sinatra をインストールしたイメージが必要と仮定します。docker search コマンドを使うことで、文字列 sinatra を含む全てのイメージを表示して、そこから適切なイメージを探せます。

```
$ docker search sinatra
NAME                                DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
training/sinatra                    Sinatra training image                          0                    [OK]
marceldegraaf/sinatra               Sinatra test app                                0
mattwarren/docker-sinatra-demo                                                      0                    [OK]
luisbebop/docker-sinatra-hello-world                                                0                    [OK]
bmorearty/handson-sinatra           handson-ruby + Sinatra for Hands on with D...   0
subwiz/sinatra                                                                      0
bmorearty/sinatra                                                                   0
...
```

コマンドを実行したら、sinatra を含む多くのイメージを表示します。表示するのは、イメージ名の一覧、スター（イメージがソーシャル上で有名かどうか測るものです。利用者はイメージを気に入れば”スター”を付けられます ）、公式（OFFICIAL）か、自動構築（AUTOMATED）といった状態です。**公式リポジトリ**とは、Docker 社のサポートよって丁寧に精査されている Docker リポジトリです。**自動構築（Automated Build（オートメーテッドビルド））**とは有効なソースコードを元に、イメージ内容が自動構築されたリポジトリです。

さて、利用可能なイメージの内容を確認します。ここでは training/sinatra イメージを使うことにします。これまで２種類のイメージ・リポジトリがありました。ubuntu のようなイメージは**ベース・イメージ**または**ルート・イメージ**と呼ばれます。このベース・イメージは Docker 社によって提供、構築、認証、サポートされています。これらは単一の単語名として表示されています。

また、training/sinatra イメージのような**ユーザ・イメージ**もあります。ユーザ・イメージとは Docker コミュニティのメンバーに属するもので、メンバーによって構築、メンテナンスされます。ユーザ・イメージは、常にユーザ名がイメージの前に付きます。この例のイメージは、training というユーザによって作成されました。

イメージの取得

適切なイメージ training/sinatra を確認したら、docker pull コマンドを使ってダウンロードできます。

```
$ docker pull training/sinatra
```

これでチームはこのイメージを使い、自身でコンテナを実行できます。

```
$ docker run -t -i training/sinatra /bin/bash
root@a8cb6ce02d85:/#
```

イメージの作成

チームでは training/sinatra イメージが有用だと分かりました。しかし、イメージを自分たちが使えるようにするには、いくつかの変更が必要です。イメージの更新や作成には２つの方法があります。

- イメージから作成したコンテナを更新し、イメージの結果をコミット
- Dockerfile（ドッカーファイル）を使って、イメージ作成の命令を指定

更新とイメージのコミット

イメージを更新するには、まず更新したいイメージからコンテナを作成する必要があります。

```
$ docker run -t -i training/sinatra /bin/bash
root@0b2616b0e5a8:/#
```

作成したコンテナ ID、ここでは 0b2616b0e5a8 をメモしておきます。このあとすぐ使います。

実行しているコンテナ内に json gem を追加しましょう。

```
root@0b2616b0e5a8:/# gem install json
```

この作業が終わったら、exit（イクジット）コマンドを使ってコンテナを終了します。

以上で自分たちが必要な変更を加えたコンテナができました。次に docker commit（コミット）コマンドを使い、イメージに対してこのコンテナのコピーを**コミット**（commit）できます。

```
$ docker commit -m "Added json gem" -a "Kate Smith" \
0b2616b0e5a8 ouruser/sinatra:v2
4f177bd27a9ff0f6dc2a830403925b5360bfe0b93d476f7fc3231110e7f71b1c
```

ここで使った docker commit コマンドの内容を確認します。２つのフラグ -m と -a を指定しています。-m フラグはコミット・メッセージを指定するもので、バージョン・コントロール・システムのようにコミットできます。-a フラグは更新を行った担当者を指定できます。

また、新しいイメージを作成する元となるコンテナを指定します。ここでは 0b2616b0e5a8（先ほど書き留めた ID）です。そして、ターゲットとなるイメージを次のように指定します。

```
ouruser/sinatra:v2
```

こちらの詳細を見ていきましょう。ouruser は新しいユーザ名であり、このイメージを書いた人です。また、イメージに対して何らかの名前も指定します。ここではオリジナルのイメージ名 sinatra をそのまま使います。最後に、イメージに対するタグ v2 を指定します。

あとは docker images コマンドを使えば、作成した新しいイメージ ouruser/sinatra が見えます。

```
$ docker images
REPOSITORY          TAG     IMAGE ID       CREATED       SIZE
training/sinatra    latest  5bc342fa0b91   10 hours ago  446.7 MB
ouruser/sinatra     v2      3c59e02ddd1a   10 hours ago  446.7 MB
ouruser/sinatra     latest  5db5f8471261   10 hours ago  446.7 MB
```

作成したイメージを使ってコンテナを作成するには、次のようにします。

```
$ docker run -t -i ouruser/sinatra:v2 /bin/bash
root@78e82f680994:/#
```

Dockerfile からイメージを構築

docker commit コマンドを使う方法は、イメージを簡単に拡張します。しかし、少々面倒なものであり、チーム内の開発プロセスでイメージを共有するのは簡単ではありません。この方法ではなく、新しいコマンド docker build を使い構築する方法や、イメージをスクラッチ（ゼロ）から作成する方法があります。

この構築コマンドを使うには **Dockerfile** を作成します。この中に Docker がどのようにしてイメージを構築するのか、命令セットを記述します。

作成するにはまず、ディレクトリと Dockerfile を作成します。

```
$ mkdir sinatra
$ cd sinatra
$ touch Dockerfile
```

Windows で Docker Machine を使っている場合は、ホスト・ディレクトリには cd で /c/Users/<ユーザ名> を指定してアクセスできるでしょう。

各々の命令で新しいイメージ層を作成します。簡単な例として、架空の開発チーム向けの Sinatra イメージを構築しましょう。

```
# ここはコメントです
FROM ubuntu:14.04
MAINTAINER Kate Smith <ksmith@example.com>
RUN apt-get update && apt-get install -y ruby ruby-dev
RUN gem install sinatra
```

Dockerfile が何をしているか調べます。それぞれの**命令**（instruction）は、**ステートメント**（statement）の前にあり、大文字で記述します。

命令 ステートメント

を使ってコメントを示せます

冒頭の **FROM 命令**は Docker に対して基となるイメージを伝えます。この例では、新しいイメージは Ubuntu 14.04 イメージを基にします。**MAINTAINER 命令**は誰がこの新しいイメージを管理するか指定します。

最後に **RUN 命令**を指定しています。RUN 命令はイメージの中で実行するコマンドを指示します。この例ではパッケージのインストールすのため、まず APT キャッシュを更新します。それから、Ruby と RubyGem をインスト

ールし、Sinatra gem をインストールします。

あとは Dockerfile を用い、docker build コマンドでイメージを構築します。

```
$ docker build -t ouruser/sinatra:v2 .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 1 : FROM ubuntu:14.04
 ---> e54ca5efa2e9
Step 2 : MAINTAINER Kate Smith <ksmith@example.com>
 ---> Using cache
 ---> 851baf55332b
Step 3 : RUN apt-get update && apt-get install -y ruby ruby-dev
 ---> Running in 3a2558904e9b
Selecting previously unselected package libasan0:amd64.
(Reading database ... 11518 files and directories currently installed.)
〜省略〜
Running hooks in /etc/ca-certificates/update.d....done.
 ---> c55c31703134
Removing intermediate container 3a2558904e9b
Step 4 : RUN gem install sinatra
 ---> Running in 6b81cb6313e5
unable to convert "\xC3" to UTF-8 in conversion from ASCII-8BIT to UTF-8 to US-ASCII for README.rdoc,
skipping
unable to convert "\xC3" to UTF-8 in conversion from ASCII-8BIT to UTF-8 to US-ASCII for README.rdoc,
skipping
Successfully installed rack-1.5.2
Successfully installed tilt-1.4.1
Successfully installed rack-protection-1.5.3
Successfully installed sinatra-1.4.5
4 gems installed
Installing ri documentation for rack-1.5.2...
Installing ri documentation for tilt-1.4.1...
Installing ri documentation for rack-protection-1.5.3...
Installing ri documentation for sinatra-1.4.5...
Installing RDoc documentation for rack-1.5.2...
Installing RDoc documentation for tilt-1.4.1...
Installing RDoc documentation for rack-protection-1.5.3...
Installing RDoc documentation for sinatra-1.4.5...
 ---> 97feabe5d2ed
Removing intermediate container 6b81cb6313e5
Successfully built 97feabe5d2ed
```

docker build コマンドで -t フラグを指定しました。ここでは新しいイメージがユーザ ouruser に属していること、リポジトリ名が sinatra 、タグを v2 に指定しました。

また、Dockerfile の場所を示すため . を使えば、現在のディレクトリにある Dockerfile の使用を指示します。

Dockerfile のパスも指定できます。

これで構築プロセスが進行します。まず Docker が行うのは構築コンテクスト（訳者注：環境の意味）のアップロードです。典型的なコンテクストとは、構築時のディレクトリです。この指定によって、Docker デーモンが実際のイメージ構築にあたり、ローカルのコンテクストをそこに入れるために必要とします。

この次は Dockerfile の命令を一行ずつ実行します。それぞれのステップで、新しいコンテナを作成し、コンテ

ナ内で命令を実行し、変更に対してコミットするのが見えるでしょう。これは先ほど見た docker commit 処理の流れです。全ての命令を実行したら、イメージ 97feabe5d2ed が残ります（扱いやすいよう ouruser/sinatra:v2 とタグ付けもしています)。そして、作業中に作成された全てのコンテナを削除し、きれいに片付けています。

127 層以上のイメージはストレージ・ドライバに関わらず作成できません。この制限が幅広く適用されるのは、イメージ全体のサイズが大きくならないよう、多くの人に最適化を促すためです。

あとは、新しいイメージからコンテナを作成できます。

```
$ docker run -t -i ouruser/sinatra:v2 /bin/bash
root@8196968dac35:/#
```

ここではイメージ作成の簡単な概要を紹介しました。他にも利用可能な命令がありますが、省略しています。ガイドの後半に読み進めれば、Dockerfile のリファレンスから、コマンドごとに更に詳細や例を参照いただけます。Dockerfile を明らかに、読めるように、管理できるようにするためには Dockerfile ベストプラクティス・ガイドもお読みください。

イメージにタグを設定

コミットまたは構築後のイメージに対しても、タグを付けられます。タグ付けには docker tag コマンドを使います。ここでは ouruser/sinatra イメージに新しいタグを付けましょう。

```
$ docker tag 5db5f8471261 ouruser/sinatra:devel
```

docker tag コマンドにはイメージ ID を使います。ここでは 5db5f8471261 です。そしてユーザ名、リポジトリ名、新しいタグを指定します。

それから、docker images コマンドを使い新しいタグを確認します。

```
$ docker images ouruser/sinatra
REPOSITORY          TAG     IMAGE ID      CREATED        SIZE
ouruser/sinatra     latest  5db5f8471261  11 hours ago   446.7 MB
ouruser/sinatra     devel   5db5f8471261  11 hours ago   446.7 MB
ouruser/sinatra     v2      5db5f8471261  11 hours ago   446.7 MB
```

イメージのダイジェスト値

v2 以上のフォーマットのイメージには、内容に対して digest（ダイジェスト）と呼ばれる識別子が割り当て可能です。作成したイメージが長期間にわたって変更がなければ、ダイジェスト値は（変更不可能なため）予想できます。イメージの digest 値を一覧表示するには、--digests フラグを使います。

```
$ docker images --digests | head
REPOSITORY                          TAG                 DIGEST
                                IMAGE ID            CREATED             SIZE
ouruser/sinatra                     latest
sha256:cbbf2f9a99b47fc460d422812b6a5adff7dfee951d8fa2e4a98caa0382cfbdbf    5db5f8471261        11
hours ago        446.7 MB
```

2.0 レジストリに対して送信（push）や取得（pull）の実行に、push か pull コマンドを使えば、その出力にイメージのダイジェスト値も含みます。このダイジェストを使っても、イメージを pull できます。

```
$ docker pull ouruser/sinatra@cbbf2f9a99b47fc460d422812b6a5adff7dfee951d8fa2e4a98caa0382cfbdbf
```

ダイジェスト値は create、run、rmi コマンドや、Dockerfile で FROM イメージの参照にも使えます。

イメージを Docker Hub に送信

イメージを構築・作成したあとは、**docker push** (プッシュ) コマンドを使って Docker Hub に送信できます。これにより、イメージを他人と共有したり、パブリックに共有したり、あるいはプライベート・リポジトリにも送信できます。

```
$ docker push ouruser/sinatra
The push refers to a repository [ouruser/sinatra] (len: 1)
Sending image list
Pushing repository ouruser/sinatra (3 tags)
. . .
```

ホストからイメージを削除

Docker ホスト上では、コンテナの削除と同じように docker rmi コマンドでイメージも削除できます。

不要になった training/sinatra イメージを削除します。

```
$ docker rmi training/sinatra
Untagged: training/sinatra:latest
Deleted: 5bc342fa0b91cabf65246837015197eecfa24b2213ed6a51a8974ae250fedd8d
Deleted: ed0fffdcdae5eb2c3a55549857a8be7fc8bc4241fb19ad714364cbfd7a56b22f
Deleted: 5c58979d73ae448df5af1d8142436d81116187a7633082650549c52c3a2418f0
```

ホストからイメージを**削除**する**時**は、どのコンテナも**対象**となるイメージを**利用**していないことを**確認**してください。

次のステップ

ここまでは、Docker コンテナ内に個々のアプリケーションを構築する方法を見てきました。次は、複数の Docker コンテナを結び付けるアプリケーション・スタック（積み重ね）の構築方法を学びましょう。

5.2.4 コンテナのネットワーク

これまでのユーザ・ガイドでは、単純なアプリケーションを構築して実行しました。また、自分でイメージの構築をしました。このセクションでは、コンテナをどのように接続するかを学びます。

コンテナ名

これまで作成してきたコンテナは、自動的にコンテナ名が作成されました。このガイドでは nostalgic_morse という古い友人のような名前でした。自動ではな、自分でもコンテナに名前を付けられます。コンテナに名前があれば、２つの便利な機能が使えます。

- コンテナに対して何らかの役割を示す名前を付ければ、簡単に覚えられます。例えば、ウェブ・アプリケーションを含むコンテナには web と名付けます。
- 名前を付ければ、それが Docker　の他コンテナから参照する時のポイントになります。後ほど、この働きをサポートする複数のコマンドを紹介します。

コンテナに名前を付けるには、 --name フラグを使います。例えば、新しく起動するコンテナを web と呼ぶには、次のように実行します。

```
$ docker run -d -P --name web training/webapp python app.py
```

docker ps コマンドで名前を確認します。

```
$ docker ps -l
CONTAINER ID  IMAGE                    COMMAND        CREATED       STATUS       PORTS
 NAMES
aed84ee21bde  training/webapp:latest python app.py  12 hours ago  Up 2 seconds 0.0.0.0:49154->5000/tcp
web
```

あるいは docker inspect を使ってもコンテナ名を確認できます。

```
$ docker inspect web
[
{
    "Id": "3ce51710b34f5d6da95e0a340d32aa2e6cf64857fb8cdb2a6c38f7c56f448143",
    "Created": "2015-10-25T22:44:17.8543671167",
    "Path": "python",
    "Args": [
        "app.py"
    ],
    "State": {
        "Status": "running",
        "Running": true,
        "Paused": false,
        "Restarting": false,
        "OOMKilled": false,
  ...
```

コンテナ名はユニークである必要があります。これが意味するのは、web と呼ばれるコンテナはただ 1 つしか使えません。もしも同じコンテナ名を再利用したいならば、新しいコンテナで名前を使う前に、古いコンテナを削除（docker rm コマンドで）しなくてはいけません。web コンテナの停止と削除をしてから、次に進みます。

```
$ docker stop web
web
$ docker rm web
web
```

コンテナをデフォルトのネットワークで起動

Docker は**ネットワーク・ドライバ**を使うことで、コンテナのネットワーク（訳者注：連結や接続するという意味の機能）をサポートします。標準では、Docker は **bridge（ブリッジ）**と **overlay（オーバレイ）**の 2 つのネットワーク・ドライバを提供します。高度な使い方として、自分でネットワーク・ドライバ・プラグインを書き、その自分のドライバでネットワークを作成することも可能です。

Docker Engine は、自動的に 3 つのデフォルト・ネットワークをインストールします。

```
$ docker network ls
NETWORK ID          NAME                DRIVER
18a2866682b8        none                null
c288470c46f6        host                host
7b369448dccb        bridge              bridge
```

bridge という名前のネットワークは特別です。特に指定しなければ、Docker は常にこのネットワーク上にコンテナを起動します。次のコマンドを試します：

```
$ docker run -itd --name=networktest ubuntu
74695c9cea6d9810718fddadc01a727a5dd3ce6a69d09752239736c030599741
```

ネットワークの調査（訳者注：network inspect コマンド）により、コンテナの IP アドレスが簡単に分かります。

```
[
    {
        "Name": "bridge",
        "Id": "f7ab26d71dbd6f557852c7156ae0574bbf62c42f539b50c8ebde0f728a253b6f",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.17.0.1/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Containers": {
            "3386a527aa08b37ea9232cbcace2d2458d49f44bb05a6b775fba7ddd40d8f92c": {
                "EndpointID": "647c12443e91faf0fd508b6edfe59c30b642abb60dfab890b4bdccee38750bc1",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            },
            "94447ca479852d29aeddca75c28f7104df3c3196d7b6d83061879e339946805c": {
                "EndpointID": "b047d090f446ac49747d3c37d63e4307be745876db7f0ceef7b311cbba615f48",
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "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": "9001"
        }
    }
]
```

コンテナを切断（disconnect）し、ネットワークからコンテナを取り外せます。切断にはネットワーク名とコンテナ名を指定します。あるいは、コンテナ ID も使えます。この例では、名前を指定する方が速いです。

```
$ docker network disconnect bridge networktest
```

コンテナをネットワークから切断しようとしても、 `bridge` という名前で組み込まれているブリッジ・ネットワークを削除できません。ネットワークとはコンテナを他のコンテナやネットワークを隔離する一般的な手法です。そのため、Docker を使い込み、自分自身でネットワークの作成も可能です。

ブリッジ・ネットワークの作成

Docker Engine はブリッジ・ネットワークとオーバレイ・ネットワークをどちらもネイティブにサポートしています。ブリッジ・ネットワークは、単一ホスト上で実行している Docker Engine でしか使えない制限があります。オーバレイ・ネットワークは複数のホストで導入でき、高度な使い方ができます。次の例は、ブリッジ・ネットワークの作成です。

```
$ docker network create -d bridge my-bridge-network
```

Docker に対して新しいネットワークで使用するブリッジ・ドライバを指定するには、 `-d` フラグを使います。このフラグを指定しなくても、同様にこの `bridge` フラグが適用されます。マシン上のネットワーク一覧を表示します。

```
$ docker network ls
NETWORK ID          NAME                DRIVER
7b369448dccb        bridge              bridge
615d565d498c        my-bridge-network   bridge
18a2866682b8        none                null
c288470c46f6        host                host
```

このネットワークを調査しても、中にはコンテナが存在しないのが分かります。

```
$ docker network inspect my-bridge-network
[
    {
        "Name": "my-bridge-network",
        "Id": "5a8afc6364bccb199540e133e63adb76a557906dd9ff82b94183fc48c40857ac",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1/16"
                }
            ]
        },
        "Containers": {},
        "Options": {}
    }
]
```

ネットワークにコンテナを追加

ウェブ・アプリケーションの構築にあたり、安全性を高めるためにネットワークを作成します。ネットワークとは、コンテナの完全な分離を提供するものと定義します。コンテナを実行する時に、コンテナをネットワークに追加できます。

PostgreSQL データベースを実行するコンテナを起動します。--net=my-bridge-netowk フラグを付けて、新しいネットワークに接続します。

```
$ docker run -d --net=my-bridge-network --name db training/postgres
```

my-bridge-network を調べたら、コンテナがアタッチ（接続）しているのが分かります。同様にコンテナを調べても、どこに接続しているのか分かります。

```
$ docker inspect --format='{{json .NetworkSettings.Networks}}'  db
{"my-bridge-network":{"NetworkID":"7d86d31b1478e7cca9ebed7e73aa0fdeec46c5ca29497431d3007d2d9e15ed99
","EndpointID":"508b170d56b2ac9e4ef86694b0a76a22dd3df1983404f7321da5649645bf7043","Gateway":"172.18.
0.1","IPAddress":"172.18.0.2","IPPrefixLen":16,"IPv6Gateway":"","GlobalIPv6Address":"","GlobalIPv6Pr
efixLen":0,"MacAddress":"02:42:ac:11:00:02"}}
```

次に進み、近くでウェブ・アプリケーションを起動します。今回は -P フラグもネットワークも指定しません。

```
$ docker run -d --name web training/webapp python app.py
```

ウェブ・アプリケーションはどのネットワーク上で実行しているのでしょうか。アプリケーションを調査したら、標準の bridge ネットワークで実行していることが分かります。

```
$ docker inspect --format='{{json .NetworkSettings.Networks}}'  web
{"bridge":{"NetworkID":"7ea29fc1412292a2d7bba362f9253545fecdfa8ce9a6e37dd10ba8bee7129812","EndpointI
D":"508b170d56b2ac9e4ef86694b0a76a22dd3df1983404f7321da5649645bf7043","Gateway":"172.17.0.1","IPAddr
ess":"172.17.0.2","IPPrefixLen":16,"IPv6Gateway":"","GlobalIPv6Address":"","GlobalIPv6PrefixLen":0,"
MacAddress":"02:42:ac:11:00:02"}}
```

次に web の IP アドレスを取得しましょう。

```
$ docker inspect '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' web
172.17.0.2
```

次は、実行中の db コンテナでシェルを開きます：

```
$ docker exec -it db bash
root@a205f0dd33b2:/# ping 172.17.0.2
ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
^C
--- 172.17.0.2 ping statistics ---
44 packets transmitted, 0 received, 100% packet loss, time 43185ms
```

少したってから CTRL-C を使って ping を終了します。ping が通らないことが分かりました。これは、２つのコンテナが異なるネットワークで実行しているからです。これを修正しましょう。次に exit を使って、コンテナか

ら出ます。

Docker のネットワーク機能は、必要に応じてコンテナに対して多くのネットワークを**接続（attach）**できます。接続は、実行中のコンテナに対しても可能です。次に、実行中の web アプリケーションを my-bridge-network に接続します。

```
$ docker network connect my-bridge-network Web
```

db アプリケーションのシェルを再び開き、ping コマンドを再度試します。今回は IP アドレスではなく、コンテナ名 web を使います。

```
$ docker exec -it db bash
root@a205f0dd33b2:/# ping web
PING web (172.19.0.3) 56(84) bytes of data.
64 bytes from web (172.19.0.3): icmp_seq=1 ttl=64 time=0.095 ms
64 bytes from web (172.19.0.3): icmp_seq=2 ttl=64 time=0.060 ms
64 bytes from web (172.19.0.3): icmp_seq=3 ttl=64 time=0.066 ms
^C
--- web ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.060/0.073/0.095/0.018 ms
```

別の IP アドレスに ping しているのが分かります。このアドレスは my-bridge-network のアドレスであり、bridge ネットワーク上のものではありません。

次のステップ

コンテナのネットワークについて学びましたので、次はコンテナにおけるデータ管理を理解します。

5.2.5 コンテナでデータを管理

これまでは基本的な Docker の概念や、Docker イメージの導入部に加え、 コンテナのネットワーク について学びました。このセクションでは、どのようにしてコンテナ内やコンテナ間でデータを管理できるかを学びます。

それでは、Docker Engine でデータを管理するための、主な手法を 2 つ見ていきます。

データ・ボリューム

データ・ボリューム（data volume）とは、1 つまたは複数のコンテナ内で、特別に設計されたディレクトリです。また、**ユニオン・ファイルシステム**（Union File System）をバイパス（迂回）するものです。データ・ボリュームは、データの保持や共有のために、複数の便利な機能を提供します。

- ボリュームはコンテナ作成時に初期化されます。コンテナのベース・イメージ上で、特定のマウント・ポイント上のデータが指定されている場合、初期化されたボリューム上に既存のデータをコピーします。
- データ・ボリュームはコンテナ間で共有・再利用できます。
- データ・ボリュームに対する変更を直接行えます。
- イメージを更新しても、データ・ボリューム上には影響ありません。
- コンテナ自身を削除しても、データ・ボリュームは残り続けます。

データ・ボリュームは、データ保持のために設計されており、コンテナのライフサイクルとは独立しています。そのため、コンテナの削除時、Docker は決して自動的にボリュームを消さないだけでなく、コンテナから参照されなくなっても”後片付け”をせず、ボリュームはそのままです。

データ・ボリュームの追加

docker create か docker run コマンドで -v フラグを使えば、コンテナにデータ・ボリュームを追加できます。-v を複数回使うことで、複数のデータ・ボリュームをマウントできます。それでは、ウェブ・アプリケーションのコンテナに対して、ボリュームを 1 つ割り当ててみましょう。

```
$ docker run -d -P --name web -v /webapp training/webapp python app.py
```

これはコンテナの中に /webapp という新しいボリュームを作成しました。

Dockerfile で VOLUME 命令を使ってもボリュームを作成できます、イメージから作成するあらゆるコンテナに対し、新しいボリュームを追加可能です。

標準では、Docker はボリュームを読み書き可能な状態でマウントします。あるいは、読み込み専用(read-only)を指定したマウントも可能です。

```
$ docker run -d -P --name web -v /opt/webapp:ro training/webapp python app.py
```

ボリュームの場所

docker inspect コマンドを使い、ホスト上でボリュームが使用中の場所を探せます。

```
$ docker inspect web
```

ボリュームに関する情報を含む、コンテナの詳細設定を表示します。出力は、おそらく次のようなものでしょう。

```
...
"Mounts": [
    {
        "Name": "fac362...80535",
        "Source": "/var/lib/docker/volumes/fac362...80535/_data",
        "Destination": "/webapp",
        "Driver": "local",
        "Mode": "",
        "RW": true,
        "Propagation": ""
    }
]
...
```

ホスト上に場所にあたるのは、上の Source（ソース） です。コンテナ内のボリューム指定は Destination（デスティネーション） です。RW の表示は、ボリュームの読み書き可能（read-write）を意味します。

データ・ボリュームとしてホスト上のディレクトリをマウント

-v フラグの使用はボリューム作成だけではありません。Docker Engine デーモンのホスト上にあるディレクトリも、コンテナにマウント可能です。

```
$ docker run -d -P --name web -v /src/webapp:/opt/webapp training/webapp python app.py
```

このコマンドはホスト側のディレクトリ/src/webapp をコンテナ内の/opt/webapp にマウントします。パス

/opt/webapp がコンテナ内のイメージに存在している場合でも、/src/webapp を重複マウントします。しかし、既存の内容は削除しません。マウントを解除したら、内容に対して再度アクセス可能となります。これは、通常の mount（マウント）コマンドと同じような動作をします。

コンテナ内のディレクトリは、/src/docs のように、常に絶対パスが必要です。**ホスト側のディレクトリ**は相対パスでも**名前**でも構いません。**ホスト側のディレクトリ**に対して絶対パスを指定したら、Docker は指定したパスを拘束マウント（bind-mount）します。この時に**名前**の値を指定したら、Docker は指定した**名前**のボリュームを作成します。

名前の値は、アルファベットの文字で開始する必要があります。具体的には、 a-z0-9 、_ （アンダースコア）、 . （ピリオド）、 - （ハイフン）です。絶対パスの場合は / （スラッシュ）で開始します。

例えば、ホスト側ディレクトリ に /foo または foo を指定可能です。/foo 値を指定したら、Docker は（ディレクトリを）拘束したマウントを作成します。foo を指定したら、Docker Engine はその名前でボリュームを作成します。

Mac または Windows 上で Docker Machine を使う場合、Docker デーモンは OS X または Windows ファイルシステム上に限定的なアクセスを行います。Docker Machine は自動的に /Users （OS X)または C:\Users (Windows）ディレクトリのマウントを試みます。つまり、OS X 上で使っているファイルやディレクトリをマウント可能です。

```
docker run -v /Users/<パス>:/<コンテナ内のパス> ...
```

Windows 上でも、同様にディレクトリのマウントが使えます。

```
docker run -v /c/Users/<パス>:/<コンテナ内のパス> ...`
```

パスには、仮想マシンのファイルシステム上にある全てのパスを指定できます。もし VirtualBox などでフォルダの共有機能を使っているのであれば、追加の設定が必要です。VirtualBox の場合は、ホスト上のフォルダを共有フォルダとして登録する必要があります。それから、Docker の -v フラグを使ってマウントできます。

ホスト上のディレクトリをマウントするのは、テストに便利かも知れません。例えば、ソースコードをコンテナの中にマウントしたとします。次にソースコードに変更を加え、アプリケーションにどのような影響があるのか、リアルタイムで確認できます。ホスト側のディレクトリは絶対パスで指定する必要があります。もしディレクトリが存在しない場合、Docker Engine のデーモンは自動的にディレクトリを作成します。このホスト・パスの自動生成機能は廃止予定です。

Docker ボリュームは、標準で読み書き可能な状態でマウントしますが、読み込み専用としてのマウントもできます。

```
$ docker run -d -P --name web -v /src/webapp:/opt/webapp:ro training/webapp python app.py
```

ここでは同じ /src/webapp ディレクトリをマウントしていますが、読み込み専用を示す ro オプションを指定しています。

mount 機能の制限により、ホスト側のソース・ディレクトリ内のサブディレクトリに移動したら、コンテナの中からホスト上のファイルシステムに移動できる場合があります。ただし、悪意を持つユーザがホストにアクセスし、ディレクトリを直接マウントする必要があります。

ホスト・ディレクトリとは、ホストに依存する性質があります。そのため、ホストディレクトリを Dockerfile でマウントできません。なぜなら、イメージの構築はポータブル（どこでも実行可能な状態の意味）であるべきだからです。全てのホスト環境でホスト・ディレクトリを使えるとは限りません。

共有ストレージをデータ・ボリュームとしてマウント

コンテナにホスト側ディレクトリをマウントできるだけではありません。いくつかの Docker **ボリューム・プラグイン**は iSCSI（アイスカジー）、NFS（エヌエフエス）、FC（ファイバチャネル）のような共有ストレージにプロビジョニングやマウントが可能です。

共有ボリュームを使う利点は、ホストに依存しない点です。つまり、あらゆるホスト上で利用可能なボリュームを扱えます。共有ストレージ・バックエンドにアクセス可能なホストと、プラグインさえインストールしておけば、コンテナがどこで動いてもボリュームを利用可能です。

`docker run` コマンドでボリューム・ドライバを使う方法は 1 つです。ボリューム・ドライバでボリュームの作成時、他の例のようにパスを指定せず、ボリューム名を指定します。

次のコマンドは `my-named-volume` という名前付きのボリュームを作成するコマンドです。作成には flocker（フロッカー） ボリューム・ドライバを使い、コンテナからは `/opt/webapp` で利用できるようにします。

```
$ docker run -d -P \
  --volume-driver=flocker \
  -v my-named-volume:/opt/webapp \
  --name web training/webapp python app.py
```

あるいは、コンテナを作成する前でも、コンテナが使うボリュームを `docker volume create` コマンドで作成できます。

次の例は `docker volume create` コマンドで `my-named-volume` ボリュームを作成します。

```
$ docker volume create -d flocker --name my-named-volume -o size=20GB
$ docker run -d -P \
  -v my-named-volume:/opt/webapp \
  --name web training/webapp python app.py
```

ボリューム・ラベル

SELinux（エスイーリナックス） のようなラベリング・システムでは、コンテナ内にマウントされたボリュームの内容に対しても、適切なラベル付けが行われます。ラベルがなければ、コンテナ内の内容物を使って実行しようとしても、セキュリティ・システムがプロセスの実行を妨げるでしょう。標準では、Docker は OS によって設定されるラベルに対して変更を加えません。

コンテナの内容物に対するラベルを変更するには、ボリュームのマウントにあたり、 `:z` または `:Z` を末尾に追加可能です（接尾辞）。これらを指定したら、Docker に対して共有ボリュームが再度ラベル付けされたものと伝えます。`z` オプションは、ボリュームの内容を複数のコンテナが共有していると Docker に伝えます。その結果、Docker は共有コンテント・ラベルとして内容をラベル付けします。`Z` オプションは、内容はプライベートで共有されるべきではない（private unshared）ラベルと Docker に伝えます。現在のコンテナのみが、プライベートに（個別に）ボリュームを利用可能です。

ホスト上のファイルをデータ・ボリュームとしてマウント

`-v` フラグはホストマシン上のディレクトリ だけ ではなく、単一のファイルに対してもマウント可能です。

```
$ docker run --rm -it -v ~/.bash_history:/.bash_history ubuntu /bin/bash
```

これは新しいコンテナ内の bash シェルを流し込むものです。コンテナを終了する時に、ホスト上の bash 履歴に対して、コンテナ内で実行したコマンドを履歴として記録します。

vi や sed --in-place など、多くのツールによる編集は、結果として i ノードを変更する場合があります。Docker v1.1.0 までは、この影響により “ sed: cannot rename ./sedKdJ9Dy: Device or resource busy ”（デバイスまたはリソースがビジー）といったエラーが表示されることがありました。マウントしたファイルを編集したい場合、親ディレクトリのマウントが最も簡単です。

データ・ボリューム・コンテナの作成とマウント

データに永続性を持たせたい場合（データを保持し続けたい場合)、例えばコンテナ間での共有や、データを保持しないコンテナから使うには、名前を付けた**データ・ボリューム・コンテナ**（Data Volume Container）を作成し、そこにデータをマウントするのが良い方法です。

ボリュームを持ち、共有するための新しい名前付きコンテナを作成しましょう。training/postgres イメージを再利用し、全てのコンテナから利用可能なレイヤ作成し、ディスク容量を節約します。

```
$ docker create -v /dbdata --name dbdata training/postgres /bin/true
```

次に、--volumes-from フラグを使い、他のコンテナから /dbdata ボリュームをマウント可能です。

```
$ docker run -d --volumes-from dbdata --name db1 training/postgres
```

あるいは、他からも。

```
$ docker run -d --volumes-from dbdata --name db2 training/postgres
```

この例では、postgres イメージには /dbdata と呼ばれるディレクトリが含まれています。そのため dbdata コンテナからボリュームをマウントする（volumes from）とは、元の postgres イメージから /dbdata が隠された状態です。この結果、dbdata コンテナからファイルを表示しているように見えます。

--volumes-from パラメータは複数回利用できます。複数のコンテナから、複数のデータボリュームを一緒に扱えます。

また、ボリュームのマウントは連鎖（chain）できます。この例では、dbdata コンテナのボリュームは db1 コンテナと db2 コンテナからマウントできるだけとは限りません。

```
$ docker run -d --name db3 --volumes-from db1 training/postgres
```

ボリュームをマウントしているコンテナを削除する場合、ここでは 1 つめの dbdata コンテナや、派生した db1 と db2 コンテナのボリュームは削除されません。ディスクからボリュームを削除したい場合は、最後までボリュームをマウントしていたコンテナで、必ず docker rm -v を実行する必要があります。この機能を使えば、コンテナ間でのデータボリュームの移行や更新を効率的に行えます。

コンテナ削除時、-v オプションでボリュームを消そうとしなくても、Docker は何ら警告を表示しません。 -v オプションに使わずにコンテナを削除した場合、ボリュームは最終的にどのコンテナからも参照されない “宙づり”(dangling) ボリュームになってしまいます。宙づりボリュームは除去が大変であり、多くのディスク容量を使用する場合もあります。このボリューム管理の改善については、現在 プルリクエスト#14214[*1] において議論中です。

*1 https://github.com/docker/docker/pull/14214

データ・ボリュームのバックアップ・修復・移行

ボリュームを使った他の便利な機能に、バックアップや修復、移行があります。これらの作業を使うには、新しいコンテナを作成する時に `--volumes-from`（ボリュームス フロム）フラグを使い、次のようにボリュームをマウントします。

```
$ docker run --volumes-from dbdata -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata
```

ここでは新しいコンテナを起動し、dbdata コンテナからボリュームをマウントします。そして、ローカルのホスト上のディレクトリを /backup としてマウントします。最終的に、dbdata ボリュームに含まれる内容をバックアップするため、 tar（ター） コマンドを使い /backup ディレクトリの中にあるファイルを backup.tar に通します。コマンドの実行が完了したら、コンテナは停止し、dbdata ボリュームのバックアップが完了します。

これで同じコンテナに修復（リストア）や、他のコンテナへの移行もできます。新しいコンテナを作成してみましょう。

```
$ docker run -v /dbdata --name dbdata2 ubuntu /bin/bash
```

それから、新しいコンテナのデータ・ボリュームにバックアップしたファイルを展開します。

```
$ docker run --volumes-from dbstore2 -v $(pwd):/backup ubuntu bash -c "cd /dbdata && \
  tar xvf /backup/backup.tar"
```

この手法を使うことで、好みのツールを用いた自動バックアップ、移行、修復が行えます。

ボリューム共有時の重要なヒント

複数のコンテナが 1 つまたは複数のデータ・ボリュームを共有できます。しかしながら、複数のコンテナが 1 つの共有ボリュームに書き込むことにより、データ破損を引き起こす場合があります。アプリケーションが共有データ・ストアに対する書き込みに対応した設計かどうか、確認してください。

データ・ボリュームは Docker ホストから直接アクセス可能です。これが意味するのは、データ・ボリュームは通常の Linux ツールから読み書き可能です。コンテナとアプリケーションが直接アクセスできることを知らないことにより、データの改竄を引き起こすことは望ましくありません。

次のステップ

これまでは、どのようにして Docker を使うのかを少々学びました。次は Docker と Docker Hub で利用可能なサービスを連携し、自動構築（Automated Build）やプライベート・リポジトリ（private repository）について学びます。

5.2.6 イメージを Docker Hub に保管

これまではローカル・ホスト上の Docker を、コマンドラインで操作する方法を学びました。 イメージを取得し、既存のイメージからコンテナを構築する方法と 自分自身でイメージを作成する方法を学びました。

次は **Docker Hub**（ハブ） を簡単に使う方法を学び、Docker のワークフローを拡張しましょう。

Docker Hub は Docker 社が管理する（パブリックな）**公開レジストリ**です。ここには、ダウンロードしてコンテナ構築に使えるイメージが置かれています。また、自動化や、ワークグループの仕組み、ウェブフック（webhook）や構築トリガ（build trigger）のようなワークフロー・ツール、一般には共有したくないイメージを保管するプライベート・リポジトリのようなプライバシー・ツールを提供します。

Docker コマンドと Docker Hub

Docker Hub が提供するサービスには、 Docker 自身が docker search（サーチ）、 pull（プル）、 login（ログイン）、 push（プッシュ）コマンドを通して接続する機能があります。ここではコマンドの働きを見ていきましょう。

アカウントの作成とログイン

例によって、Docker Hub を使い始めるには（未作成であれば）アカウントを作成し、ログインします。アカウントの作成は Docker Hub 上で[*1] 行えます。

```
$ docker login
```

これでコミットしたイメージを、自分の Docker Hub リポジトリ上に、アップロード（push）できます。

自分の認証に関する情報は、自分のホームディレクトリの /.docker/config.json に保管されます。

イメージの検索

（ブラウザの）検索インターフェースかコマンドライン・インターフェースを通して、Docker Hub レジストリを検索できます。イメージの検索は、イメージ名、ユーザ名、説明文に対して可能です。

```
$ docker search centos
NAME            DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
centos          The official build of CentOS                    1223      [OK]
tianon/centos   CentOS 5 and 6, created using rinse instea...   33
...
```

ここでは centos と tianon/centos という、２つの結果が表示されました。後者は tianon という名前のユーザによる公開リポジトリです。１つめの結果 centos とは明確に異なるリポジトリです。１つめの centos は、**公式リポジトリ（Official Repository）**としての信頼されるべきトップ・レベルの名前空間です。文字列 / により、イメージ名とユーザのリポジトリ名を区別します。

欲しいイメージが見つかれば、 docker pull <イメージ名> によってダウンロードできます。

```
$ docker pull centos
Using default tag: latest
latest: Pulling from library/centos
f1b10cd84249: Pull complete
c852f6d61e65: Pull complete
7322fbe74aa5: Pull complete
Digest: sha256:90305c9112250c7e3746425477f1c4ef112b03b4abe78c612e092037bfecc3b7
Status: Downloaded newer image for centos:latest
```

これで、この入手したイメージをから、コンテナを実行可能です。

バージョンの指定と最新版

docker pull centos の実行は、docker pull centos:latest の実行と同等です。イメージを取得するにあたり、標

*1 https://hub.docker.com/account/signup/

準の**最新**（**latest**）イメージをダウンロードするのではなく、より適切なイメージを正確に指定可能です。

例えば、centos のバージョン 5 を取得するには、docker pull centos:centos5 を使います。この例では、 centos リポジトリにおける centos のバージョンを、タグ centos5 でラベル付けしたイメージを指定しています。

リポジトリにおいて現在利用可能なタグの一覧を確認するには、 Docker Hub 上のレジストリをご覧ください。

Docker Hub への貢献

誰でも Docker Hub レジストリから公開イメージを取得（pull）できるよう設定可能です。自分のレジストリでイメージを共有したい場合は、まず登録が必要です。

Docker Hub にリポジトリの送信

リポジトリを対象のレジストリに**送信**（**push**）するためには、イメージに名前を付けるか、コンテナにイメージ名を付けてコミットする必要があります。

それからこのリポジトリを、レジストリが表す名前やタグで送信できます。

```
$ docker push yourname/newimage
```

対象のイメージをアップロードしたら、あなたの同僚やコミュニティにおいても利用可能になります。

Docker Hub の機能

それでは、Docker Hub のいくつかの機能について、詳細を見ていきましょう。

プライベート・リポジトリ

イメージを一般公開せず、誰とも共有したくない場合があります。そのような時は Docker Hub のプライベート・リポジトリが利用できます。サインアップや料金プランは、 サイト[*1] をご覧ください。

組織とチーム

プライベート・リポジトリの便利な機能の 1 つは、組織やチームにおける特定メンバーのみとの共有です。Docker Hub 上で**組織**（**organization**）を作り、同僚と協力しながらプライベート・リポジトリの管理が可能です。組織の作成や管理方法については Docker Hub の章をご覧ください。

自動構築

自動構築（**Automated Build**）とは、GItHub[*2] や Bitbucket[*3] を更新したら、Docker Hub が直接イメージの構築や更新をします。これは、選択した GitHub か Bitbucket リポジトリに対するコミットをきっかけ（フック）とします。コミットをプッシュ（push）したのをトリガとして、イメージを構築・更新します。

自動構築のセットアップ

1. Docker Hub アカウントを作成してログインします。
2. ［ Linked Accounts & Services ］（アカウントとサービスのリンク）から自分の GitHub もしくは Bitbucket アカウントをリンクします。
3. 自動構築の設定をします。
4. 選択した GitHub もしくは Bitbucket プロジェクト上で、構築内容を Dockerfile にまとめます。
5. 必要があれば構築時のブランチを指定します（デフォルトは master ブランチです）。

*1 https://registry.hub.docker.com/plans/

*2 https://www.github.com/

*3 http://bitbucket.com/

6. 自動構築名を指定します。
7. 構築時に追加するオプションの Docker タグを指定します。
8. `Dockerfile` の場所を指定します。デフォルトは `/` です。

自動構築の設定を有効化しておけば、ビルドをトリガとして数分後に自動構築が開始します。自動ビルドの状態は Docker Hub レジストリ上で見られます。GitHub や Bitbucket リポジトリの同期が終わるまで、自動ビルドを無効化できません。

リポジトリの自動構築状態や出力を確認したい場合は、自分のリポジトリ一覧ページに移動し、対象のリポジトリ名をクリックします。自動構築が有効な場合は、リポジトリ名の下に“automated build”と表示されます。リポジトリの詳細ページに移動し、”Build details”タブをクリックしたら、Docker Hub 上における構築状態や、全ての構築トリガが表示されます。

自動構築が完了したら、無効化や設定の削除が可能になります。ここで注意すべきは、`docker push` コマンドを使って送信しても、自動構築を行わない点です。自動構築の管理対象は、あくまでも GitHub と Bitbucket リポジトリに対してコードをコミットした時のみです。

リポジトリごとに複数の自動構築設定や、特定の Dockerfile や Git ブランチの指定も可能です。

構築のトリガ

Docker Hub の URL を経由しても、自動構築のトリガにできます。これにより、イメージを必要に応じて自動的に再構築することが可能です。

ウェブフック

ウェブフック（webhook）とは、リポジトリに対して設定します。トリガとなるのは、イメージに対するイベントの発生や、更新されたイメージがリポジトリに送信された時です。ウェブフックは特定の URL JSON ペイロードで指定でき、イメージが送信（push）されると適用されます。

ウェブフックの詳細 については、Docker Hub の章をご覧ください。

次のステップ

さぁ Docker を使いましょう！

5.3 イメージの活用

5.3.1 Dockerfile を書くベスト・プラクティス

Docker は Dockerfile（ドッカーファイル）の命令を読み込み、自動的にイメージを構築できます。これはテキストファイルであり、特定のイメージを構築するために必要な全ての命令が入っています。Dockerfile は個別の命令セットを使い、特定の形式で記述します。基本は Dockerfile リファレンスで学べます。新しく Dockerfile を書こうとしているのであれば、そのリファレンスから出発しましょう。

このドキュメントは Docker 社が推奨するベストプラクティスや手法を扱っており、Docker コミュニティが簡単かつ効率的に Dockerfile を作成できるようにします。この推奨方法に従うことを強く勧めます（実際、公式イメージを作成するには、これらのプラクティスに従う必要があります)。

多くのベストプラクティスや推奨する手法は、buildpack-deps[*1] の Dockerfile でご覧いただけます。

Dockerfile の命令に関するより詳細な説明は、Dockerfil リファレンスのページをご覧ください。

一般的なガイドラインとアドバイス

コンテナはエフェメラルであるべき

Dockerfile で定義されたイメージを使って作成するコンテナは、可能ならば**エフェメラル（短命；ephemeral）**にすべきです。私たちの「エフェメラル」とは、停止・破棄可能であり、明らかに最小のセットアップで構築して使えることを意味します。

.dockerignore ファイルの利用

ほとんどの場合、空のディレクトリに個々の Dockerfile を置くのがベストです。そうしておけば、そのディレクトリには Dockerfile が構築に必要なファイルだけ追加します。構築のパフォーマンスを高めるには、.dockerignore（ドッカーイグノア）ファイルを作成し、対象ディレクトリ上のファイルやディレクトリを除外できます。このファイルの除外パターンは .gitignore（ギットイグノア）ファイルに似ています。ファイルを作成するには、.dockerignore のリファレンスをご覧ください。

不要なパッケージをインストールしない

複雑さ、依存関係、ファイルサイズ、構築階数をそれぞれ減らすために、余分ないし不必要な「入れた方が良いだろう」というパッケージは、インストールを避けるべきです。例えば、データベース・イメージであればテキストエディタは不要でしょう。

コンテナごとに１つのプロセスだけ実行

ほとんどの場合、１つのコンテナの中で１つのプロセスだけ実行すべきです。アプリケーションを複数のコンテナに分離することは、水平スケールやコンテナの再利用を簡単にします。サービスとサービスに依存関係がある場合は、コンテナのリンク機能を使います。

レイヤの数を最小に

Dockerfile の読みやすさと、使用するイメージレイヤ数の最小化は、この両者のバランスを見つける必要があります。戦略的に注意深くレイヤ数をお使いください。

*1 https://github.com/docker-library/buildpack-deps/blob/master/jessie/Dockerfile

複数行の引数

可能であれば常に複数行の引数をアルファベット順にします。これにより、パッケージが重複しないようにし、あるいはリストの更新を容易にします。また、プルリクエストの読み込みとレビューをより簡単にします。バックスラッシュ(\ ) の前に空白を追加するのも、同じく役立つでしょう。

以下は buildpack-deps イメージの記述例です。

```
RUN apt-get update && apt-get install -y \
  bzr \
  cvs \
  git \
  mercurial \
  subversion
```

構築キャッシュ

Docker イメージ構築のプロセスとは、Dockerfile で指定した各命令を順番に実行していきます。それぞれの命令で、新しい（あるいは重複した）イメージを作成するのではなく、Docker は既存イメージのキャッシュがないか確認します。もしキャッシュを全く使いたくなければ、 docker build コマンドで --no-cache=true（ノーキャッシュ トゥルー） オプションを使います。

一方で非常に重要なのは、Docker でキャッシュ機能を使うにあたり、マシンイメージを探す時に、何を行い何を行わないかの理解です。Docker の基本的な処理の概要は、次の通りです。

- 開始にあたり、ベース・イメージが既にキャッシュにあれば、次の命令を対象のベース・イメージから派生した全ての子イメージと比較します。同じ命令があれば構築にそのイメージを使います。もし同じ命令がなければ、キャッシュを無効化します。
- ほとんどの場合は、Dockerfile 命令と子イメージの単純な比較で十分です。しかし、命令によっては更に検査や追加検査が必要になります。
- ADD と COPY 命令は、イメージに含まれるファイルが検査され、各ファイルのチェックサムを計算します。ファイルの最終編集・最終アクセス時間は、チェックサムに影響ありません。キャッシュを探し、既存のイメージのチェックサムと比較します。内容やメタデータのようにファイルに変更があれば、キャッシュを無効化します。
- ADD と COPY コマンドだけでなく、キャッシュのチェックにおいて、キャッシュが一致すると思われるコンテナ内のファイル状態を確認しません。例えば、RUN apt-get -y update コマンドによってコンテナ内のファイルに変更を加えたとしても、キャッシュの有無に影響を与えません。この場合、コマンドの文字列自身が一致するかどうかしか見ないためです。
- キャッシュを無効化すると、以降の Dockerfile 命令ではキャッシュは使われず、新しいイメージを生成します。

Dockerfile 命令

以下では、Dockerfile で様々な命令を使うにあたり、推奨するベストな方法が分かるでしょう。

FROM（フロム）

可能であれば、自分のイメージの元として現在の公式リポジトリを使います。私たちは Debian イメージ[1] を推奨します。これは、非常にしっかりと管理されており、ディストリビューションの中でも小さくなるよう（現在は 150 MB 以下に）維持されているからです。

*1 https://hub.docker.com/_/debian/

RUN（ラン）

常に Dockerfile をより読みやすく、理解しやすく、メンテナンスしやすくします。長ければ分割するか、複雑な RUN 命令はバックスラッシュを使い複数行に分割します。

おそらく RUN の最も一般的な利用例は apt-get アプリケーションです。RUN apt-get コマンドはパッケージをインストールしますので、気を付けるべきいくつかの注意点があります。

まず、RUN apt-get update や dist-upgrade を避けるべきでしょう。ベース・イメージに含まれる「必須」パッケージの多くが、権限を持たないコンテナの内部で更新されないためです。もし、ベース・イメージに含まれるパッケージが時代遅れになっていれば、イメージのメンテナに連絡すべきでしょう。例えば、foo という特定のパッケージを知っていて、それを更新する必要があるのであれば、自動的に更新するために apt-get install -y foo を使います。

RUN apt-get update と apt-get install は常に同じ RUN 命令文で連結します。以下は実行例です。

```
RUN apt-get update && apt-get install -y \
    package-bar \
    package-baz \
    package-foo
```

RUN 命令文で apt-get update だけを使うとキャッシュ問題を引き起こし、その後の apt-get install 命令が失敗します。例えば、次のように Dockerfile を記述したとします。

```
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y curl
```

イメージを構築後、Docker は全てのレイヤをキャッシュします。次に、別のパッケージを追加す apt-get install を編集したとします。

```
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y curl nginx
```

Docker は冒頭からファイルを読み込み、命令の変更を認識したら、前回作成したキャッシュを再利用します。つまり、apt-get update は決して実行されず、キャッシュされたバージョンを使います。これは apt-get update を実行していないため、古いバージョンの curl と nginx パッケージを取得する恐れがあります。

そこで、Dockerfile でインストールする場合は RUN apt-get update && apt-get install -y を指定し、最新バージョンのパッケージを、追加の記述や手動作業なく利用できます。

```
RUN apt-get update && apt-get install -y \
    package-bar \
    package-baz \
    package-foo=1.3.*
```

パッケージのバージョン指定は、何をキャッシュしているか気にせずに、特定バージョンを取得した上での構築を強制します。このテクニックにより、必要なパッケージが、予期しない変更によって引き起こされる失敗を減らします。

以下は　丁寧に練られた RUN 命令であり、推奨する apt-get の使い方の全てを示すものです。

```
RUN apt-get update && apt-get install -y \
    aufs-tools \
    automake \
    build-essential \
    curl \
    dpkg-sig \
    libcap-dev \
    libsqlite3-dev \
    lxc=1.0* \
    mercurial \
    reprepro \
    ruby1.9.1 \
    ruby1.9.1-dev \
    s3cmd=1.1.* \
 && apt-get clean \
 && rm -rf /var/lib/apt/lists/*
```

s3cmd の命令行は、バージョン 1.1.*を指定します。従来のイメージが古いバージョンを使っていたとしても、新しいイメージは apt-get update でキャッシュを使いません。そのため、確実に新しいバージョンをインストールします。他の各行はパッケージのリストであり、パッケージの重複という間違いをさせないためです。

更に、apt キャッシュをクリーンにし、/var/lib/apt/lits を削除すると、イメージのサイズを減らします。 RUN 命令は apt-get update から開始しますので、 apt-get install は常に新しいパッケージをインストールします。

CMD（コマンド）

CMD 命令は、イメージに含まれるソフトウェアの実行と、その引数のために使うべきです。また、CMD は常に CMD [“実行ファイル”, “パラメータ 1”, “パラメータ 2”…] のような形式で使うべきです。そのため、イメージがサービス向け（Apache、Rails 等）であれば、CMD ["apache2","-DFOREGROUND"] のようにすべきでしょう。実際に、あらゆるサービスのベースとなるイメージで、この命令形式が推奨されます。

その他の多くの場合、CMD はインタラクティブなシェル（bash、python、perl 等）で使われます。例えば、CMD ["perl", "-de0"]、 CMD ["python"]、 CMD [“php”, “-a”] です。この利用形式が意味するのは、 docker run -it python のように実行したら、そのコマンドを使いやすいシェル上に落とし込み、すぐに使えるようにします。 また、あなたとあなたの想定ユーザが ENTRYPOINT の動作に慣れていないなら、 CMD [“パラメータ”, “パラメータ”] の形式のように、CMD を ENTRYPOINT と一緒に使うべきではないでしょう。

EXPOSE（エクスポーズ）

EXPOSE 命令は、コンテナが接続用にリッスンするポートを指定します。そのため、アプリケーションが一般的に使う、あるいは、伝統的なポートを使うべきです。例えば、Apache ウェブ・サーバのイメージは EXPOSE 80 を使い、MongoDB を含むイメージであれば EXPOSE 27017 を使うでしょう。

外部からアクセスするためには、ユーザが docker run 実行時にフラグを指定し、特定のポートを任意のポートに割り当てられます。コンテナのリンク機能を使えば、Docker はコンテナがソースをたどれるよう、環境変数を提供します（例：MYSQL_PORT_3306_TCP）。

ENV（エンブ）

新しいソフトウェアを簡単に実行するため、コンテナにインストールされているソフトウェアの PATH 環境変数を ENV を使って更新できます。例えば、ENV PATH /usr/local/nginx/bin:$PATH は CMD ["nginx"]を動作するようにします。

また、ENV 命令は PostgreSQL の PGDATA のような、コンテナ化されたサービスが必要な環境変数を指定するのにも便利です。

あとは、ENV は一般的に使うバージョン番号の指定にも使えます。そのため、バージョンを特定したメンテナンスを次のように簡単にします。

```
ENV PG_MAJOR 9.3
ENV PG_VERSION 9.3.4
RUN curl -SL http://example.com/postgres-$PG_VERSION.tar.xz | tar -xJC /usr/src/postgress && …
ENV PATH /usr/local/postgres-$PG_MAJOR/bin:$PATH
```

プログラムにおける恒常的な変数と似ています。しかし、(ハード・コーディングされた値とは違い) この手法は ENV 命令の指定で、コンテナ内のソフトウェアのバージョンを自動的に選べるようにします。

ADD と COPY

ADD と COPY の機能は似ていますが、一般的には COPY が望ましいと言われています。これは、ADD よりも機能が明確なためです。 COPY はローカルファイルをコンテナの中にコピーするという、基本的な機能しかサポートしていません。一方の ADD は複数の機能 (ローカル上での tar アーカイブ展開や、リモート URL のサポート) を持ち、一見では処理内容が分かりません (訳者注：ファイルや URL に何が含まれているか確認できないためです)。したがって ADD のベストな使い方は、ローカルの tar ファイルをイメージに自動展開 (ADD rootfs.tar.xz /) する用途です。

内容によっては、一度にファイルを取り込むよりも、Dockerfile の複数ステップで COPY することもあるでしょう。これにより、何らかのファイルが変更された所だけ、キャッシュが無効化されます (ステップを強制的に再実行します)。

例：

```
COPY requirements.txt /tmp/
RUN pip install /tmp/requirements.txt
COPY . /tmp/
```

RUN ステップはキャッシュ無効化の影響が少なくなるよう、 COPY . /tmp/ の前に入れるべきでしょう。

イメージ・サイズの問題のため、ADD でリモート URL 上のパッケージを取得するのは可能な限り避けてください。その代わりに curl や wget を使うべきです。この方法であれば、展開後に不要となったファイルを削除でき、イメージに余分なレイヤを増やしません。例えば、次のような記述は避けるべきです。

```
ADD http://example.com/big.tar.xz /usr/src/things/
RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things
RUN make -C /usr/src/things all
```

そのかわり、次のように記述します。

```
RUN mkdir -p /usr/src/things \
    && curl -SL http://example.com/big.tar.xz \
    | tar -xJC /usr/src/things \
    && make -C /usr/src/things all
```

他のアイテム (ファイルやディレクトリ) は ADD の自動展開機能を必要としませんので、常に COPY を使うべきです。

ENTRYPOINT

ENTRYPOINT のベストな使い方は、イメージにおけるメインコマンドの設定です。これによりイメージを指定したコマンドを通して実行します (そして、CMD がデフォルトのフラグとして使われます)。

コマンドライン・ツール s3cmd のイメージを例にしてみましょう。

```
ENTRYPOINT ["s3cmd"]
CMD ["--help"]
```

このイメージを使って次のように実行したら、コマンドのヘルプを表示します。

```
$ docker run s3cmd
```

あるいは、適切なパラメータを指定したら、コマンドを実行します。

```
$ docker run s3cmd ls s3://mybucket
```

イメージ名が先ほどの命令で指定したコマンドのバイナリも兼ねているため、使いやすくなります。

また、ENTRYPOINT 命令は役に立つスクリプトの組みあわせにも利用できます。そのため、ツールを使うために複数のステップが必要になるかもしれない場合も、先ほどのコマンドと似たような方法が使えます。

例えば、 Postgres 公式イメージは次のスクリプトを ENTRYPOINT に使っています。

```
#!/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 "$@"
```

このスクリプトは exec Bash コマンドをコンテナの PID 1 アプリケーションとして実行します。これにより、コンテナに対して送信される Unix シグナルは、アプリケーションが受信します。詳細は ENTRYPOINT のヘルプをご覧ください。

ヘルパー・スクリプトをコンテナの中にコピーし、コンテナ開始時の ENTRYPOINT で実行します。

```
COPY ./docker-entrypoint.sh /
ENTRYPOINT ["/docker-entrypoint.sh"]
```

このスクリプトは Postgres とユーザとの対話に利用できます。例えば、単純な postgres の起動に使えます。

```
$ docker run postgres
```

あるいは、PostgreSQL 実行時、サーバに対してパラメータを渡せます。

```
$ docker run postgres postgres --help
```

または、Bash のように全く異なったツールのためにも利用可能です。

```
$ docker run --rm -it postgres bash
```

VOLUME（ボリューム）

VOLUME 命令はデータベース・ストレージ領域、設定用ストレージ、Docker コンテナによって作成されるファイルやフォルダの公開のために使います。イメージにおいて変わりやすい場所・ユーザによって便利な場所としてVOLUME の利用が強く推奨されます。

USER（ユーザ）

USER を使えばサービスを特権ユーザで実行せずに、 root 以外のユーザで実行できます。利用するには Dockerfile で RUN groupadd -r postgres && useradd -r -g postgres postgres のようにユーザとグループを作成します。

イメージ内で得られるユーザとグループは UID/GID に依存しないため、イメージの構築に関係なく次の UID/GID が割り当てられます。そのため、これが問題になるのであれば、UID/GID を明確に割り当ててください。

TTY やシグナル送信を使わないつもりであれば、sudo のインストールや使用を避けたると良いでしょう。使うことで引き起こされる問題の解決は大変だからです。もし、どうしても sudo のような機能が必要であれば（例：root としてデーモンを初期化しますが、実行は root 以外で行いたい時)、 「 gosu（ゴースー）[*1] 」を利用できます。

あとは、レイヤの複雑さを減らすため、 USER を頻繁に切り替えるべきではありません。

WORKDIR（ワークディレクトリ）

明確さと信頼性のため、常に WORKDIR からの絶対パスを使うべきです。また、 WORKDIR を RUN cd ... && 何らかの処理 のように増殖する命令の代わり使うことで、より読みやすく、トラブルシュートしやすく、維持しやすくします。

ONBUILD（オンビルド）

ONBULID 命令は、Dockerfile による構築後に実行します。ONBUILD は FROM から現在に至るあらゆる子イメージで実行できます。 ONBUILD コマンドは親の Dockerfile が子 Dockerfile を指定する命令としても考えられます。

Docker は ONBUILD コマンドを処理する前に、あらゆる子 Dockerfil 命令を実行します。

ONBUILD は FROM で指定したイメージを作ったあと、更にイメージを作るのに便利です。例えば、言語スタック・イメージで ONBUILD を使えば、Dockerfile 内のソフトウェアは特定の言語環境を使えるようになります。これは Ruby の ONBUILD でも見られます 。

ONBUILD によって構築されるイメージは、異なったタグを指定すべきです。例：ruby:1.9-onbuild や ruby:2.0-onbuild 。

ONBUILD で ADD や COPY を使う時は注意してください。追加された新しいリソースが新しいイメージ上で見つからなければ、「onbuild」イメージに破壊的な失敗をもたらします。先ほどお勧めしたように、別々のタグを付けておけば、Dockerfile の書き手が選べるようになります。

*1 https://github.com/tianon/gosu

公式リポジトリの例

模範的な Dockerfile の例をご覧ください。

- Go - https://hub.docker.com/_/golang/
- Perl - https://hub.docker.com/_/perl/
- Hy - https://hub.docker.com/_/hylang/
- Rails - https://hub.docker.com/_/rails/

5.3.2 ベース・イメージの作成

自分自分で ベース・イメージ を作りたいですか？ 素晴らしいです！
Linux ディストリビューションによっては、パッケージ化の対象により、重度に依存する手順を踏みます。以下の例では、皆さんが新しいイメージのコントリビュート（貢献）にあたり、プル・リクエストの送信を勇気づけるでしょう。

イメージ全体を tar で作成

一般に、皆さんが作業で使っているマシン上で動作しているディストリビューションを使い、ベース・イメージのパッケージを作りたいと考えるでしょう。その場合、Debian の Debootstrap[*1] のようなツールを使えば、Ubuntu イメージを使わずに構築も可能です。
Ubuntu ベース・イメージの作成するには、次のように簡単にできます。

```
$ sudo debootstrap raring raring > /dev/null
$ sudo tar -C raring -c . | docker import - raring
a29c15f1bf7a
$ docker run raring cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=13.04
DISTRIB_CODENAME=raring
DISTRIB_DESCRIPTION="Ubuntu 13.04"
```

Docker の GitHub リポジトリ上に、ベース・イメージ作成のためのサンプル・スクリプトがあります。

- BusyBox
  https://github.com/docker/docker/blob/master/contrib/mkimage-busybox.sh
- CentOS / Scientific Linux CERN (SLC) 、 Debian/Ubuntu 、 CentOS/RHEL/SLC/等
  https://github.com/docker/docker/blob/master/contrib/mkimage-rinse.sh
- Debian / Ubuntu
  https://github.com/docker/docker/blob/master/contrib/mkimage-debootstrap.sh

スクラッチからベース・イメージを作成

Docker が準備した最小イメージ scratch を、コンテナの構築開始点として使えます。 scratch “イメージ”が意味するのは、自分が Dockerfile 上のコマンドによって作成するイメージの、その最初のファイルシステム層にあた

*1 https://wiki.debian.org/Debootstrap

ります。

scratch は Docker Hub 上のリポジトリからは見えません。そのため、取得 (pull) の実行や、イメージを scratch という名前でタグ付けできません。そのかわり、Dockerfile で参照可能です。例えば、scratch を使って最小コンテナを作成するには、次のようにします。

```
FROM scratch
ADD hello /
CMD ["/hello"]
```

これはチュートリアルで使用する hello-world イメージを作成する例です。テストしたい場合は、 このイメージのリポジトリ[*1] から複製できます。

更に詳細について

Dockerfile を書くにあたり、多くの情報が手助けになるでしょう。

- Dockerfile リファレンスのセクションには、 利用可能な命令の全ガイドがあります。
- 作成した Dockerfile を、よりきれいに、読みやすく、メンテナンスしやすいように、 ベスト・プラクティス・ガイドが役立ちます。
- もし自分で新しい公式リポジトリを作成するのが目標であれば、Docker の 公式リポジトリについてのページをお読みください。

5.3.3 イメージの管理

Docker エンジンにあるクライアントを使い、コマンドライン上でイメージの作成や、構築プロセスに渡せます。これらのイメージをコンテナとしての使用や、他人が使えるように公開も可能です。作成したイメージの保管や、使いたいイメージの検索、あるいは、他人が公開しているのを使いたい場合、これらは全てイメージ管理の範囲です。

このセクションでは、イメージ管理に関する Docker の主な機能概要や、ツールについて扱います。

Docker Hub

Docker Hub が持っている役割は、ユーザ・アカウント、イメージ、パブリック名前空間などに関する情報の集約です。

- ウェブ UI
- メタ・データの保管（コメント、スター数、公開リポジトリの一覧）認証サービス
- トークン化

Docker 社が運用・管理しているのは Docker Hub だけです。この公開 Hub は、ほとんどの個人や小さな会社にとって便利です。

Docker レジストリと Docker トラステッド・レジストリ

Docker **レジストリ（registry）**は Docker エコシステムの構成要素（コンポーネント）です。レジストリは保管と**コンテント（content; 内容）**の配送システムであり、Docker イメージの名前を保持し、異なったタグのバージ

*1 https://github.com/docker-library/hello-world

ョンを利用できます。例えば、イメージ `distribution/registry` は、タグ `2.0` と `latest` を持っています。ユーザは `docker pull myregistry.com/stevvooe/batman:voice` のように、 `docker push`（送信）と `pull`（取得）コマンドを使いレジストリと接続します。

Docker Hub はハブ（中継点）のように自身のレジストリを持ち、Docker 社が運用・管理しています。しかしながら、レジストリを得るには別の方法があります。Docker **トラステッド・レジストリ（Trusted Registry）** プロダクトを購入しますと、自社ネットワーク上で実行できるようになります。あるいは、Docker レジストリのコンポーネントを使い、プライベート・レジストリを構築することもできます。レジストリの使い方についての情報は、 Docker レジストリの章をご覧ください。

コンテント・トラスト

ネットワーク・システム上でデータ転送時には、信頼性が懸念事項の中心です。特にインターネットのような信頼できない環境を経由する時、とりわけ重要なのが、システムが操作する全データの安全性と、提供者を保証することです。Docker を使えば、イメージ（データ）をリポジトリに送信・受信できます。コンテント・トラストは、レジストリがどの経路をたどっても、全てのデータの安全性と提供者の両方を保証するものです。

コンテント・トラスト（Content Trust）は、現時点ではパブリックな Docker Hub の利用者だけが使えます。現時点では Docker トラステッド・レジストリやプライベート・レジストリでは利用できません。

5.4 Docker ストレージ・ドライバ

Docker はストレージを管理するドライバ技術を頼り、イメージとコンテナを実行するため相互に連携して動きます。

新しい Docker コンテナを使う前に、まずイメージ、コンテナ、ストレージ・ドライバの理解をお読みください。重要な概念と技術に関する説明がありますので、ストレージ・ドライバがどのような動作をするのか理解する手助けになるでしょう。

謝辞

Docker ストレージ・ドライバの基となる大部分は、ゲスト開発者の Nigel Poulton 氏によって書かれたものです。Docker 社自身の Jérôme Petazzoni も僅かながら手助けを行いました。Nigel 氏は IT トレーニングビデオ の作成、 In Tech We Trust ポッドキャスト に多くの時間を費やし、大部分は Twitter 上で過ごします。

5.4.1 イメージ、コンテナ、ストレージ・ドライバの理解

ストレージ・ドライバを効率的に使うには、Docker がどのようにイメージを構築・保管するかの理解が欠かせません。そして、これらのイメージがコンテナでどのように使われているかの理解が必要です。最後に、イメージとコンテナの両方を操作するための技術に対する、簡単な紹介をします。

イメージとレイヤ

Docker イメージは読み込み専用（read-only）レイヤのセットです。それぞれのレイヤが層（スタック）として積み重なり、1つに統合された形に見えます。この1番めの層を**ベース・イメージ（base image）**と呼び、他の全てのレイヤは、このベース・イメージのレイヤ上に積み重なります。次の図は、 Ubuntu 15:04 イメージが4つのイメージ・レイヤを組みあわせて構成されているのが分かります。

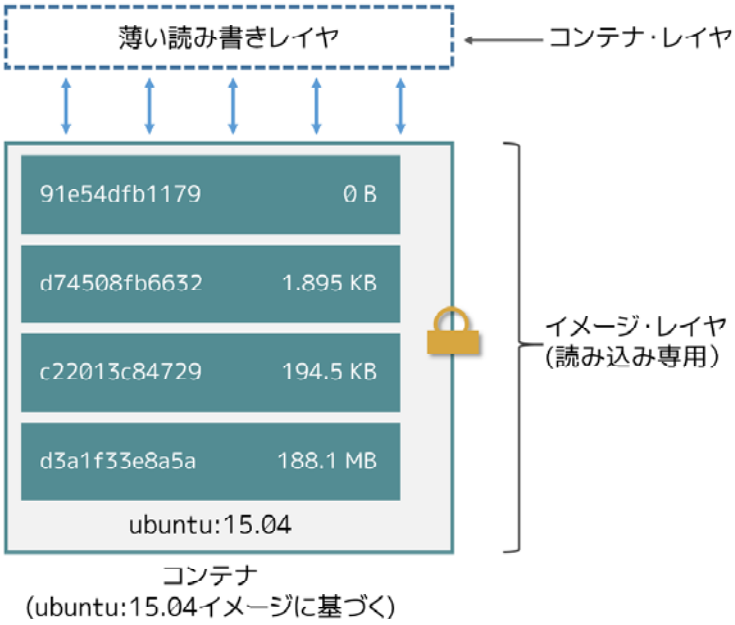
イメージ

Docker ストレージ・ドライバは、これらレイヤを積み重ねて単一に見えるようにする役割があります。

コンテナ内部に変更を加えた時を考えます。例えば、Ubuntu 15.04 イメージ上に新しくファイルを追加したら、下にあるイメージ層の上に、新しいレイヤを追加します。この変更は、新しく追加したファイルを含む新しいレイヤを作成します。各イメージ・レイヤは自身の UUID（universal unique identifier）を持っており、下の方にあるイメージの上に、連続したイメージ・レイヤを構築します。

コンテナ（ストレージの内容を含みます）は Docker イメージと薄い書き込み可能なレイヤとを連結したものです。この書き込み可能なレイヤは一番上にあり、**コンテナ・レイヤ（container layer）**と呼ばれます。以下の図は

ubuntu 15.04 イメージの実行状態です。

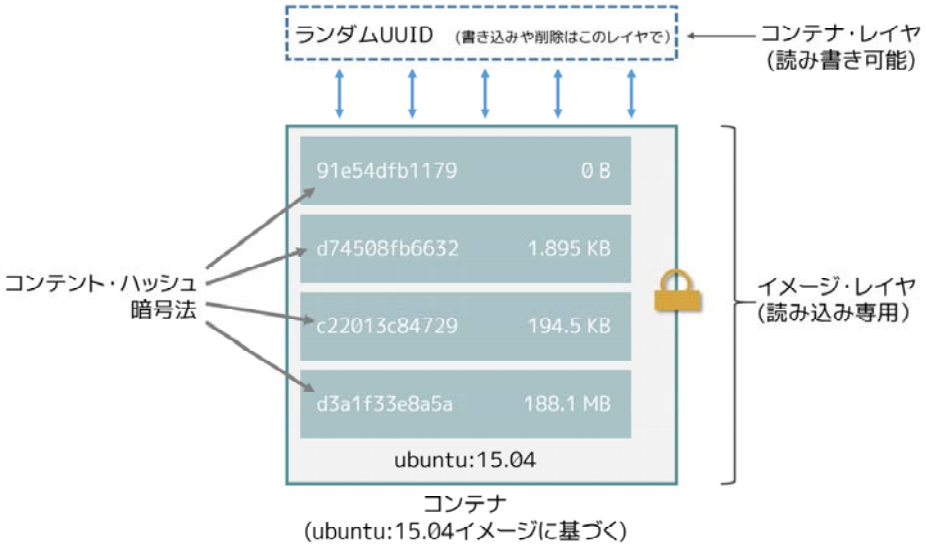


連想ストレージ

Docker 1.10 は、新しい**連想（コンテント・アドレッサブル；content adressable）ストレージ・モデル**を導入しました。これはイメージとレイヤをディスクで扱うための、全く新しい手法です。従来のイメージとレイヤのデータは、ランダムに生成した UUID を使って保管・参照していました。新しいモデルでは、これを安全な**コンテント・ハッシュ（content hash）**[*1] に置き換えます。

新しいモデルはセキュリティを改善します。ID の重複を防ぐ機能を持っており、pull・push・load・save 操作を実施後の、データ保証を完全なものとします。また、同時に構築していなくても、多くイメージが各レイヤを自由に共有可能にもなりました。

次の図は、従来バージョンの図を更新したものです。Docker 1.10 で実装された変更をハイライトしています。



こちらにある通り、まだコンテナ ID がランダムな UUID であるのに対して、全てのイメージ・レイヤの ID は

*1 訳者注：内容を証明するハッシュ値の意味合いです。

暗号化ハッシュです。

新しいモデルに関しては、いくつかの注意点があります。

1. 既存イメージの移行
2. イメージとレイヤのファイルシステム構造

既存イメージとは、以前のバージョンの Docker で作成、あるいは取得したものです。これらは新しいモデルで使う前に、変換が必要です。以降時には、新しい安全なチェックサムを計算します。この計算は更新した Docker デーモンを初回起動時、自動的に行われます。移行が終わったら、全てのイメージとタグが新しい安全な ID に更新されます。

移行は自動的かつ透過的に行われますが、多くの計算を必要とします。つまり、イメージ・データが大量にあれば、時間がかかることを意味します。移行している間、Docker デーモンは他のリクエストに応答しません。

新しいイメージへの移行を、Docker デーモンをアップグレードする前に行えるツールがあります。つまり、移行に時間をかけないので、停止時間の発生を避けられます。また、既存のイメージを手動で移行できますので、最新バージョンの Docker が既に動いている環境への移行も可能です。

Docker 社が提供しているデータの移行用ツールは、コンテナとして実行できます。ダウンロードは https://github.com/docker/v1.10-migrator/releases からできます。

「migrator（マイグレータ）」イメージの実行中は、Docker ホストのデータ・ディレクトリをコンテナに対して公開する必要があります。Docker データを置く場所がデフォルトであれば、コマンドラインでコンテナを実行するには、次のようにします。

```
$ sudo docker run --rm -v /var/lib/docker:/var/lib/docker docker/v1.10-migrator
```

devicemapper ストレージ・ドライバを使っている場合は、 --privileged オプションを使ってコンテナがストレージ・デバイスにアクセスできるようにする必要があります。

移行例

以下の例は、Docker デーモンのホスト・バージョンが 1.9.1 で、AUFS ストレージ・ドライバを使っている環境を移行します。Docker ホストは t2.micro AWS EC2 インスタンス上で動いており、1 vCPU 、1GB メモリ、8GB の SSD EBS ボリュームを持っています。Docker のデータ・ディレクトリ（/var/lib/docker）は 2GB の容量を使っています。

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
jenkins             latest              285c9f0f9d3d        17 hours ago        708.5 MB
mysql               latest              d39c3fa09ced        8 days ago          360.3 MB
mongo               latest              a74137af4532        13 days ago         317.4 MB
postgres            latest              9aae83d4127f        13 days ago         270.7 MB
redis               latest              8bccd73928d9        2 weeks ago         151.3 MB
centos              latest              c8a648134623        4 weeks ago         196.6 MB
ubuntu              15.04               c8be1ac8145a        7 weeks ago         131.3 MB

$ sudo du -hs /var/lib/docker
2.0G    /var/lib/docker

$ time docker run --rm -v /var/lib/docker:/var/lib/docker docker/v1.10-migrator
Unable to find image 'docker/v1.10-migrator:latest' locally
latest: Pulling from docker/v1.10-migrator
ed1f33c5883d: Pull complete
```

```
b3ca410aa2c1: Pull complete
2b9c6ed9099e: Pull complete
dce7e318b173: Pull complete
Digest: sha256:bd2b245d5d22dd94ec4a8417a9b81bb5e90b171031c6e216484db3fe300c2097
Status: Downloaded newer image for docker/v1.10-migrator:latest
time="2016-01-27T12:31:06Z" level=debug msg="Assembling tar data for
01e70da302a553ba13485ad020a0d77dbb47575a31c4f48221137bb08f45878d from
/var/lib/docker/aufs/diff/01e70da302a553ba13485ad020a0d77dbb47575a31c4f48221137bb08f45878d"
time="2016-01-27T12:31:06Z" level=debug msg="Assembling tar data for
07ac220aeeef9febf1ac16a9d1a4eff7ef3c8cbf5ed0be6b6f4c35952ed7920d from
/var/lib/docker/aufs/diff/07ac220aeeef9febf1ac16a9d1a4eff7ef3c8cbf5ed0be6b6f4c35952ed7920d"
<snip>
time="2016-01-27T12:32:00Z" level=debug msg="layer
dbacfa057b30b1feaf15937c28bd8ca0d6c634fc311ccc35bd8d56d017595d5b took 10.80 seconds"

real    0m59.583s
user    0m0.046s
sys     0m0.008s
```

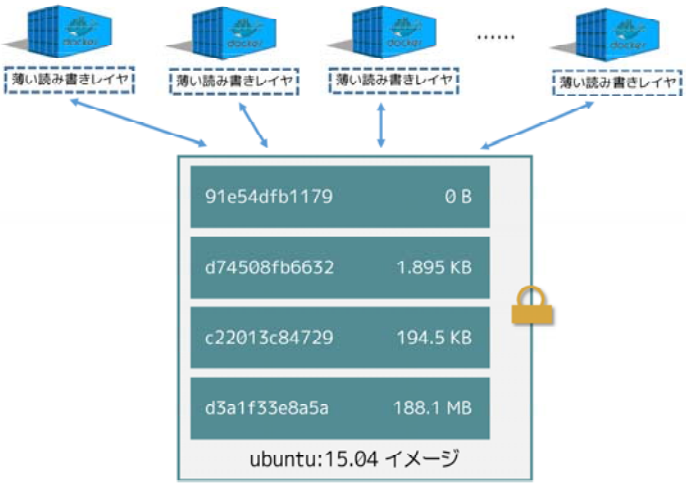
Unix の time コマンドを docker run コマンドより前に付け、処理時間を計測します。表示されているように、2GB の容量を消費している 7 つのディスク・イメージの移行に、おおよそ 1 分かかっています。しかし、これには docker/v1.10-migrator イメージの取得（約 3.5 秒）も含みます。同じ処理を m4.10xlarge EC2 インスタンス、40 VCPU 、160GB のメモリ、8GB の provisioned IOPS EBS ボリュームであれば、次のような結果になります。

```
real    0m9.871s
user    0m0.094s
sys     0m0.021s
```

以上の結果から、処理時間は移行をするマシンのハードウェア性能に影響を受けることが分かります。

コンテナとレイヤ

それぞれのコンテナは、自分自身で書き込み可能なレイヤを持ちますので、全てのデータは対象のコンテナレイヤに保管します。つまり、複数のコンテナが根底にあるイメージを共有アクセスすることができ、それぞれのコンテナ自身がデータをも管理できるのを意味します。次の図は複数のコンテナが同じ Ubuntu 15.04 イメージを共有しています。



ストレージ・ドライバは、イメージ・レイヤと書き込み可能なコンテナ・レイヤの両方を有効化・管理する責任があります。ストレージ・ドライバは様々な方法で処理をします。Docker イメージとコンテナ管理という２つの重要な技術の裏側にあるのは、積み上げ可能なイメージ・レイヤとコポイー・オン・ライトです。

コピー・オン・ライト方式

共有とはリソース最適化のための良い手法です。人々はこれを日常生活通で無意識に行っています。例えば双子の Jane と Joseph が代数学のクラスを受ける時、回数や先生が違っても、同じ教科書を相互に共有できます。ある日、Jane が本のページ 11 にある宿題を片付けようとしています。その時 Jane はページ 11 をコピーし、宿題を終えたら、そのコピーを提出します。Jane はページ 11 のコピーに対する変更を加えただけであり、オリジナルの教科書には手を加えていません。

コピー・オン・ライト（copy-on-write、cow）とは、共有とコピーのストラテジ（訳者注：方針、戦略の意味、ここでは方式と訳します）に似ています。この方式は、システム・プロセスが自分自身でデータのコピーを持つより、同一インスタンス上にあるデータ共有を必要とします。書き込む必要があるプロセスのみが、データのコピーにアクセスできます。その他のプロセスは、オリジナルのデータを使い続けられます。

Docker はコピー・オン・ライト技術をイメージとコンテナの両方に使います。この CoW 方式はイメージのディスク使用量とコンテナ実行時のパフォーマンスの両方を最適化します。次のセクションでは、イメージとコンテナの共有とコピーにおいて、コピー・オン・ライトがどのように動作してるのかを見てきます。

共有を促進する小さなイメージ

このセクションではイメージ・レイヤとコピー・オン・ライト技術（copy-on-write）を見ていきます。全てのイメージとコンテナ・レイヤは Docker ホスト上の ローカル・ストレージ領域 に存在し、ストレージ・ドライバによって管理されます。Linux をベースとする Docker ホストでは、通常は `/var/lib/docker/` 以下です。

イメージを取得・送信する `docker pull` と `docker push` 命令の実行時、Docker クライアントはイメージ・レイヤについて報告します。以下のコマンドは、 Docker Hub から `ubuntu:15.04` Docker イメージを取得（pull）しています。

```
$ docker pull ubuntu:15.04
15.04: Pulling from library/ubuntu
1ba8ac955b97: Pull complete
f157c4e5ede7: Pull complete
0b7e98f84c4c: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:5e279a9df07990286cce22e1b0f5b0490629ca6d187698746ae5e28e604a640e
Status: Downloaded newer image for ubuntu:15.04
```

この出力から、このコマンドが実際には４つのイメージ・レイヤを取得したのが分かります。上記のそれぞれの行が、イメージとその UUID か暗号化ハッシュです。これら４つのレイヤの組みあわせにより、 ubuntu:15.04 Docker イメージを作り上げています。

これらの各レイヤは、Docker ホスト上のローカル・ストレージ領域に保管します。

Docker バージョン 1.10 未満までは、各レイヤをイメージ・レイヤ ID と同じ名前のディレクトリに格納していました。しかし、Docker バージョン 1.10 移行では、イメージを取得してもこのようになりません。例えば、Docker Engine バージョン 1.9.1 が動いているホスト上で、Docker Hub からイメージをダウンロードするコマンドを実行した結果です。

```
$  docker pull ubuntu:15.04
15.04: Pulling from library/ubuntu
47984b517ca9: Pull complete
df6e891a3ea9: Pull complete
```

```
e65155041eed: Pull complete
c8be1ac8145a: Pull complete
Digest: sha256:5e279a9df07990286cce22e1b0f5b0490629ca6d187698746ae5e28e604a640e
Status: Downloaded newer image for ubuntu:15.04

$ ls /var/lib/docker/aufs/layers
47984b517ca9ca0312aced5c9698753ffa964c2015f2a5f18e5efa9848cf30e2
c8be1ac8145a6e59a55667f573883749ad66eaeef92b4df17e5ea1260e2d7356
df6e891a3ea9cdce2a388a2cf1b1711629557454fd120abd5be6d32329a0e0ac
e65155041eed7ec58dea78d90286048055ca75d41ea893c7246e794389ecf203
```

4 つのディレクトリが、イメージをダウンロードしたレイヤの ID と一致しているのが分かるでしょう。これと同じ処理を Docker Engine バージョン 1.10 上で行いましょう。

```
$ docker pull ubuntu:15.04
15.04: Pulling from library/ubuntu
1ba8ac955b97: Pull complete
f157c4e5ede7: Pull complete
0b7e98f84c4c: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:5e279a9df07990286cce22e1b0f5b0490629ca6d187698746ae5e28e604a640e
Status: Downloaded newer image for ubuntu:15.04

$ ls /var/lib/docker/aufs/layers/
1d6674ff835b10f76e354806e16b950f91a191d3b471236609ab13a930275e24
5dbb0cbe0148cf447b9464a358c1587be586058d9a4c9ce079320265e2bb94e7
bef7199f2ed8e86fa4ada1309cfad3089e0542fec8894690529e4c04a7ca2d73
ebf814eccfe98f2704660ca1d844e4348db3b5ccc637eb905d4818fbfb00a06a
```

先ほどの結果とは異なり、4 つのディレクトリは取得したイメージ・レイヤ ID と対応しません。

このように、バージョン 1.10 前後ではイメージの管理に違いがあります。しかし全ての Docker バージョンにおいて、イメージはレイヤを共有できます。例えば、イメージを pull（取得）する時、既に取得済みの同じイメージ・レイヤがあれば、Docker は状況を認識してイメージを共有します。そして、ローカルに存在しないイメージのみ取得します。2 つめ以降の pull は、共通イメージ・レイヤにある 2 つのイメージを共有しています。

これで、自分で実例を試して理解できるでしょう。 ubuntu:15.04 イメージを使うため、まずは取得（pull）し、変更を加え、その変更に基づく新しいイメージを構築します。この作業を行う方法の 1 つが、 Dockerfile と docker build コマンドを使う方法です。

1. 空のディレクトリに、Dockerfile を作成します。 ubuntu:15.04 イメージの指定から記述します。

```
FROM ubuntu:15.04
```

2.「newfile」という名称の新規ファイルを、イメージの /tmp ディレクトリに作成します。ファイル内には「Hello world」の文字も入れます。

作業が終われば、 Dockerfile は次の 2 行になっています。

```
FROM ubuntu:15.04

RUN echo "Hello world" > /tmp/newfile
```

3. ファイルを保存して閉じます。

4. ターミナルから、作成した Dockerfile と同じディレクトリ上に移動し、以下のコマンドを実行します。

```
$ docker build -t changed-ubuntu .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM ubuntu:15.04
 ---> 3f7bcee56709
Step 2 : RUN echo "Hello world" > /tmp/newfile
 ---> Running in d14acd6fad4e
 ---> 94e6b7d2c720
Removing intermediate container d14acd6fad4e
Successfully built 94e6b7d2c720
```

上記のコマンドの末尾にあるピリオド（.）は重要です。これは docker build コマンドに対して、現在の作業用ディレクトリを構築時のコンテクスト（内容物）に含めると伝えるものです。

この結果から、新しいイメージのイメージ ID が 94e6b7d2c720 だと分かります。

5. docker images コマンドを実行します。

6. Docker ホスト上のローカル・ストレージ領域に、新しい changed-ubuntu イメージが作成されているかどうかを確認します。

```
REPOSITORY        TAG       IMAGE ID        CREATED             SIZE
changed-ubuntu    latest    03b964f68d06    33 seconds ago      131.4 MB
ubuntu            15.04     013f3d01d247    6 weeks ago         131.3 MB
```

7. docker history コマンドを実行します。

8. 新しい changed-ubuntu イメージが何のイメージによって作成されたか分かります。

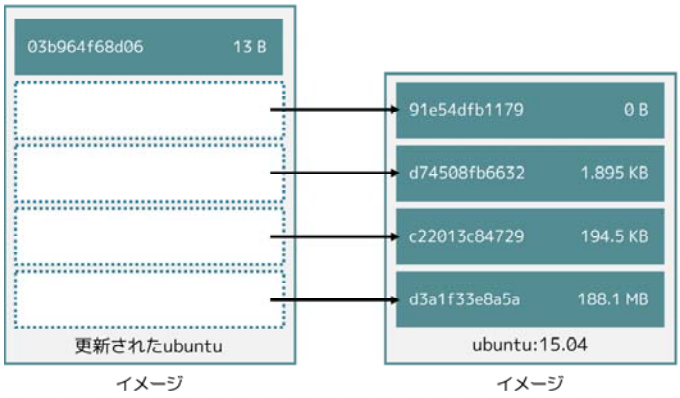
```
$ docker history changed-ubuntu
IMAGE               CREATED             CREATED BY                                      SIZE
COMMENT
94e6b7d2c720        2 minutes ago       /bin/sh -c echo "Hello world" > /tmp/newfile     12 B
3f7bcee56709        6 weeks ago         /bin/sh -c #(nop) CMD ["/bin/bash"]              0 B
<missing>           6 weeks ago         /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/    1.879 kB
<missing>           6 weeks ago         /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic    701 B
<missing>           6 weeks ago         /bin/sh -c #(nop) ADD file:8e4943cd86e9b2ca13    131.3 MB
```

docker history の出力結果から、新しい 94e6b7d2c720 イメージ・レイヤが一番上にあることが分かります。03b964f68d06 レイヤとは、先ほどの Dockerfile で echo "Hello world" > /tmp/newfile コマンドでファイルを追加したものだと分かります。そして、4 つのイメージ・レイヤは、先ほど ubuntu:15.04 イメージを構築する時に使ったレイヤと一致していることが分かります。

Docker 1.10 で導入された連想ストレージ・モデル（content addressable storage model）下では、イメージの履歴データは各イメージ・レイヤの設定ファイル上に保存されません。これからは、イメージ全体に関連する、単一の設定ファイル上の文字列に保管されます。これにより、docker history コマンドを実行したら、いくつかのイメージ・レイヤは「missing」（行方不明）と表示されるでしょう。しかしこれは通常の動作であり、無視して構いません。

これらのイメージを フラット・イメージ（flat images）として読んでいるのを耳にしているかもしれません。

新しい changed-ubuntu イメージは各レイヤのコピーを自分自身で持っていないことに注意してください。下図にあるように、ubuntu:15.04 イメージの下にある４つのレイヤを、新しいイメージでも共有しているのです。

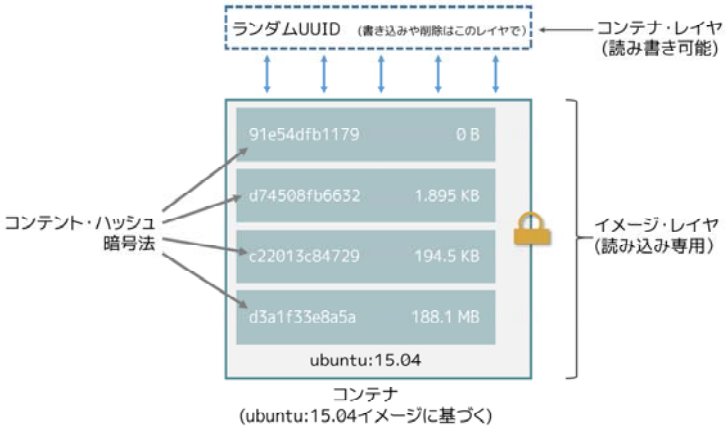


また、docker history コマンドは各イメージ・レイヤのサイズも表示します。94e6b7d2c720 は 12 バイトのディスク容量です。つまり、changed-ubuntu は Docker ホスト上の 12 バイトのディスク容量しか消費しません。これは、94e6b7d2c720 よりも下層のレイヤにあたるものが Docker ホスト上に存在しており、これらは changed-ubuntu イメージとして共有されているからです。

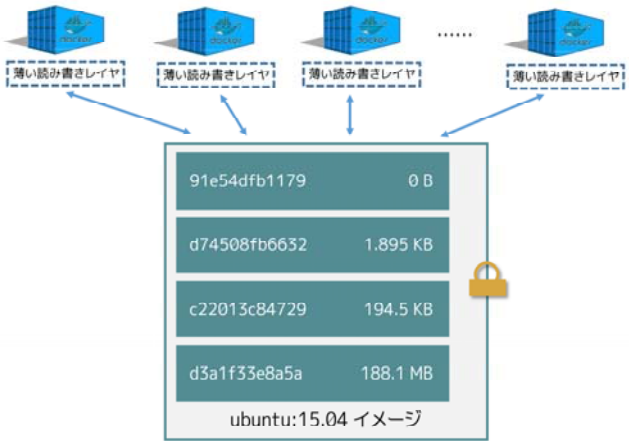
このイメージ・レイヤの共有により、Docker イメージとコンテナの領域を効率的に扱えます。

コンテナを効率的にコピー

先ほど学んだように、Docker イメージのコンテナとは、書き込み可能なコンテナ・レイヤを追加したものです。以下の図は ubuntu:15.04 をコンテナのベース・レイヤと下層レイヤを表示しています。



コンテナに対する全ての書き込みは、書き込み可能なコンテナ・レイヤに保管されます。他のレイヤは読み込み専用（read-only、RO）のイメージ・レイヤであり、変更できません。つまり、複数のコンテナが下層にある１つのイメージを安全に共有できます。以下の図は、複数のコンテナが ubuntu:15.04 イメージのコピーを共有しています。各コンテナは自分自身で読み書き可能なレイヤを持っていますが、どれもが ubuntu:15.04 イメージという単一のインスタンス（イメージ）を共有しています。



コンテナの中で書き込み作業が発生したら、Docker はストレージ・ドライバでコピー・オン・ライト処理を実行します。この処理はストレージ・ドライバに依存します。AUFS と OverlayFS ストレージ・ドライバは、コピー・オン・ライト処理を、おおよそ次のように行います。

- レイヤ上のファイルが更新されていないか確認します。まずこの手順が新しいレイヤに対して行われ、以降は１つ１つのベースになったレイヤをたどります。
- ファイルに対して初めての処理が始まると「コピー開始」(copy-up) をします。「コピー開始」とは、コンテナ自身が持つ薄い書き込み可能なレイヤから、ファイルをコピーすることです。
- コンテナの薄い書き込み可能なレイヤに ファイル を コピー してから、(そのファイルに) 変更を加えます。

BTFS、ZFS、その他のドライバは、コピー・オン・ライトを異なった方法で処理します。これらのドライバの手法については、後述するそれぞれの詳細説明をご覧ください。

たくさんのデータが書き込まれたコンテナは、何もしないコンテナに比べて多くのディスク容量を消費します。これは書き込み操作の発生によって、コンテナの薄い書き込み可能なレイヤ上に、更に新しい領域を消費するためです。もしコンテナが多くのデータを使う必要があるのであれば、データ・ボリュームを使うこともできます。

コピー開始処理は、顕著な性能のオーバーヘッド（処理時間の増加）を招きます。このオーバーヘッドは、利用するストレージ・ドライバによって異なります。しかし、大きなファイル、多くのレイヤ、深いディレクトリ・ツリーが顕著な影響を与えます。幸いにも、これらの処理が行われるのは、何らかのファイルに対する変更が初めて行われた時だけです。同じファイルに対する変更が再度行われても、コピー開始処理は行われず、コンテナ・レイヤ上に既にコピーしてあるファイルに対してのみ変更を加えます。

先ほど構築した changed-ubuntu イメージの元となる５つのコンテナに対し、何が起こっているのか見ていきましょう。

1. Docker ホスト上のターミナルで、 次のように docker run コマンドを５回実行します。

```
$ docker run -dit changed-ubuntu bash
75bab0d54f3cf193cfdc3a86483466363f442fba30859f7dcd1b816b6ede82d4
$ docker run -dit changed-ubuntu bash
9280e777d109e2eb4b13ab211553516124a3d4d4280a0edfc7abf75c59024d47
$ docker run -dit changed-ubuntu bash
a651680bd6c2ef64902e154eeb8a064b85c9abf08ac46f922ad8dfc11bb5cd8a
$ docker run -dit changed-ubuntu bash
8eb24b3b2d246f225b24f2fca39625aaad71689c392a7b552b78baf264647373
$ docker run -dit changed-ubuntu bash
```

```
0ad25d06bdf6fca0dedc38301b2aff7478b3e1ce3d1acd676573bba57cb1cfef
```

これは changed-ubuntu イメージを元に、５つのコンテナを起動します。コンテナを作成したことで、Docker は書き込みレイヤを追加し、そこにランダムな UUID を割り当てます。この値は、 docker run コマンドを実行して返ってきたものです。

2. docker ps コマンドを実行し、５つのコンテナが実行中なのを確認します。

```
$ docker ps
CONTAINER ID    IMAGE             COMMAND    CREATED              STATUS              PORTS    NAMES
0ad25d06bdf6    changed-ubuntu    "bash"     About a minute ago   Up About a minute            stoic_ptolemy
8eb24b3b2d24    changed-ubuntu    "bash"     About a minute ago   Up About a minute            pensive_bartik
a651680bd6c2    changed-ubuntu    "bash"     2 minutes ago        Up 2 minutes                 hopeful_turing
9280e777d109    changed-ubuntu    "bash"     2 minutes ago        Up 2 minutes                 backstabbing_mahavira
75bab0d54f3c    changed-ubuntu    "bash"     2 minutes ago        Up 2 minutes                 boring_pasteur
```

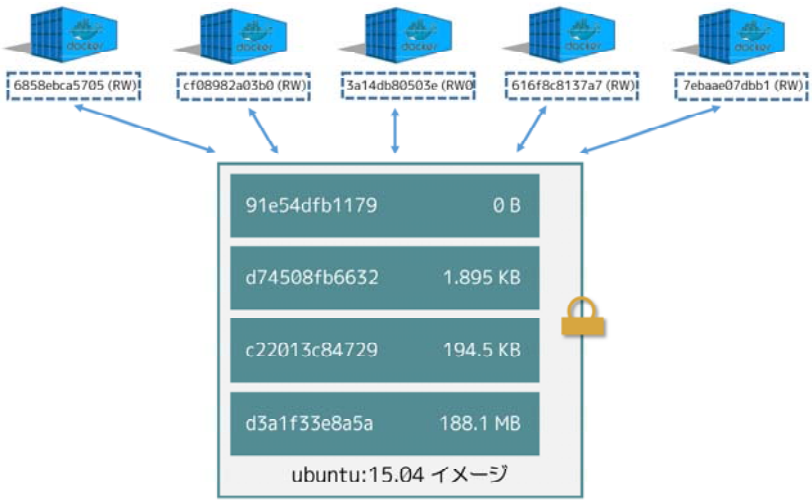
上記の結果から、changed-ubuntu イメージを全て共有する５つのコンテナが実行中だと分かります。それぞれのコンテナ ID は各コンテナ作成時の UUID から与えられています。

3. ローカル・ストレージ領域のコンテナ一覧を表示します。

```
$ sudo ls containers
0ad25d06bdf6fca0dedc38301b2aff7478b3e1ce3d1acd676573bba57cb1cfef
9280e777d109e2eb4b13ab211553516124a3d4d4280a0edfc7abf75c59024d47
75bab0d54f3cf193cfdc3a86483466363f442fba30859f7dcd1b816b6ede82d4
a651680bd6c2ef64902e154eeb8a064b85c9abf08ac46f922ad8dfc11bb5cd8a
8eb24b3b2d246f225b24f2fca39625aaad71689c392a7b552b78baf264647373
```

（翻訳者注：上記コマンドは、 /var/lib/docker ディレクトリで実行してください。）

Docker のコピー・オン・ライト方式により、コンテナによるディスク容量の消費を減らすだけではなく、コンテナ起動時の時間も短縮します。起動時に、Docker はコンテナごとに薄い書き込み可能なレイヤを作成します。次の図は changed-ubuntu イメージの読み込み専用のコピーを、５つのコンテナで共有しています。



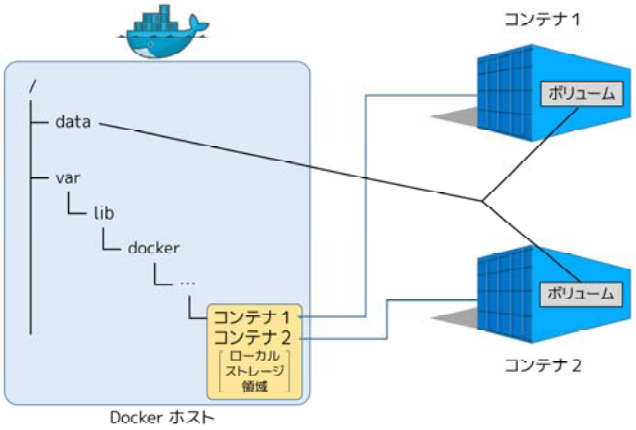
もし新しいコンテナを開始する度に元になるイメージ・レイヤ全体をコピーするのであれば、コンテナの起動時間とディスク使用量が著しく増えてしまうでしょう。

データ・ボリュームとストレージ・ドライバ

コンテナを削除したら、コンテナに対して書き込まれたあらゆるデータを削除します。しかし、 **データ・ボリューム（data volume）**の保存内容は、コンテナと一緒に削除しません。

データ・ボリュームとは、コンテナが直接マウントするディレクトリまたはファイルであり、Docker ホストのファイルシステム上に存在します。データ・ボリュームはストレージ・ドライバが管理しません。データ・ボリュームに対する読み書きはストレージ・ドライバをバイパス（迂回）し、ホスト上の本来の速度で処理されます。コンテナ内に複数のデータ・ボリュームをマウントできます。１つまたは複数のデータ・ボリュームを、複数のコンテナで共有もできます。

以下の図は、１つの Docker ホストから２つのコンテナを実行しているものです。Docker ホストのローカル・ストレージ領域（/var/lib/docker/...）の中に、それぞれのコンテナに対して割り当てられた領域が存在しています。また、Docker ホスト上の /data に位置する共有データ・ボリュームもあります。このディレクトリは両方のコンテナからマウントされます。



データ・ボリュームは Docker ホスト上のローカル・ストレージ領域の外に存在しており、ストレージ・ドライバの管理から独立して離れています。コンテナを削除したとしても、Docker ホスト上の共有データ・ボリュームに保管されたデータに対して、何ら影響はありません。

データ・ボリュームに関する更に詳しい情報は、 「コンテナでデータを管理する」セクションをご覧ください。

5.4.2 ストレージ・ドライバの選択

このセクションは Docker のストレージ・ドライバ機能を説明します。Docker がサポートしているストレージ・ドライバの一覧と、ドライバ管理に関連する基本的なコマンドを扱います。ページの最後では、ストレージ・ドライバの選び方のガイドを提供します。

なお、このセクションは、既にストレージ・ドライバ技術を理解している読者を想定しています。

交換可能なストレージ・ドライバ構造

Docker は交換可能な（pluggable）ストレージ・ドライバ構造を持っています。そのため、自分の環境や使い方に応じて、ベストなストレージ・ドライバを「プラグイン」（接続）として選べます。これが柔軟さをもたらします。Docker の各ストレージ・ドライバは、 Linux ファイルシステムやボリューム・マネージャの技術に基づいています。そのうえ、各ストレージ・ドライバはイメージ・レイヤとコンテナ・レイヤの管理を、各々の独自手法により自由に管理方法を実装できます。つまり、同じストレージ・ドライバであっても、異なった状況では性能が良くなるのを意味します。

どのドライバがベストかを決めたら、Docker デーモンの起動時にドライバを指定するだけです。Docker デーモンは対象のストレージ・ドライバを使って起動します。そして、デーモン・インスタンスによって作成される全て

のコンテナは、全てその同じストレージ・ドライバを使っています。次の表はサポートされているストレージ・ドライバ技術とドライバ名です。

技術	ストレージ・ドライバ名
OverlayFs	overlay
AUFS	aufs
Btrfs	btrfs
Device Mapper	devicemapper
VFS	vfs
ZFS	zfs

デーモンで何のストレージ・ドライバが設定されているかを確認するには、 docker info コマンドを使います。

```
$ docker info
Containers: 0
Images: 0
Storage Driver: overlay
 Backing Filesystem: extfs
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.19.0-15-generic
Operating System: Ubuntu 15.04
... 以下の出力は省略...
```

info サブコマンドで分かるのは、Docker デーモンが overlay ストレージ・ドライバを使い、Backing Filesystem[*1] を extfs の値にしています。 extfs の値は、 overlay ストレージ・ドライバを、既存の（ext）ファイルシステム上で行うという意味があります。

自分がどのストレージ・ドライバを使うか選ぶにあたり、Docker ホストのローカル・ストレージ領域で使おうとするファイルシステムに一部依存します。いくつかのストレージ・ドライバは、異なったファイルシステム技術の上でも操作できます。しかしながら、特定のストレージ・ドライバは特定のファイルシステム技術を必要とします。例えば、 btrfs ストレージ・ドライバを使うには btrfs ファイルシステム技術を使う必要があります。以下の表は、各ストレージ・ドライバが、それぞれホスト上の何のファイルシステム技術をサポートしているかの一覧です。

ストレージ・ドライバ	ファイルシステムと一致する必要
OverlayFs	なし
AUFS	なし
Btrfs	あり
Device Mapper	なし
VFS	なし
ZFS	あり

ストレージ・ドライバを設定するには、docker daemon コマンドで --storage-driver=<名前> オプションを使うか、あるいは、/etc/default/docker ファイル中の DOCKER_OPTS 行を編集します。

*1 訳者注：背後のファイルシステム、すなわち、ホスト OS 上のファイルシステムです。

以下のコマンドは Docker デーモンの起動時、 docker daemon コマンドで devicemapper ストレージ・ドライバを指定しています。

```
$ docker daemon --storage-driver=devicemapper &

$ docker info
Containers: 0
Images: 0
Storage Driver: devicemapper
 Pool Name: docker-252:0-147544-pool
 Pool Blocksize: 65.54 kB
 Backing Filesystem: extfs
 Data file: /dev/loop0
 Metadata file: /dev/loop1
 Data Space Used: 1.821 GB
 Data Space Total: 107.4 GB
 Data Space Available: 3.174 GB
 Metadata Space Used: 1.479 MB
 Metadata Space Total: 2.147 GB
 Metadata Space Available: 2.146 GB
 Udev Sync Supported: true
 Deferred Removal Enabled: false
 Data loop file: /var/lib/docker/devicemapper/devicemapper/data
 Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata
 Library Version: 1.02.90 (2014-09-01)
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.19.0-15-generic
Operating System: Ubuntu 15.04
<出力を省略>
```

ストレージ・ドライバの選択は、コンテナ化されたアプリケーションの性能に影響を与えます。そのために大切になるのは、どのようなストレージ・ドライバのオプションが利用可能かを理解し、アプリケーションに対する正しい選択をすることです。このセクションの後半では、適切なドライバを選ぶためのアドバイスを扱います。

共有ストレージ・システムとストレージ・ドライバ

多くの商用システムでは、SAN や NAS アレイのような共有ストレージ・システムをストレージ容量に使います。性能や安定性を向上させるためだけでなく、プロビジョニング・冗長化・圧縮など、高度な機能を提供します。

Docker ストレージ・ドライバとデータ・ボリュームは、共有ストレージ・システムが提供するストレージ上でも操作可能です。そのため、これらの提供されるシステムによって、Docker の性能と可用性を増大できます。しかしながら、 Docker はこれら基盤システムとは統合できません。

各ストレージ・ドライバは Linux ファイルシステムやボリューム・マネージャを基盤としているのを覚えておいてください。自分の共有ストレージ・システム上で、ストレージ・ドライバ（ファイルシステムやボリューム）を操作するベスト・プラクティスを理解してください。例えば、ZFS ストレージ・ドライバを XYZ 共有ストレージ・システム上で使うのであれば、XYZ 共有ストレージ・システム上の ZFS ファイルシステムの操作のベストプラクティスを理解すべきです。

望ましいストレージ・ドライバの選択

ストレージ・ドライバの選択には、複数の要素が影響を与えます。しかしながら、２つの事実を覚え続けなくてはけません。

- 全てのユースケースに適用できるドライバは存在しない
- ストレージ・ドライバは常に改良・進化し続けている

これらの要素を頭に入れつつ、以下で扱うポイントと表が、検討にあたっての材料になるでしょう。

安定性

Docker の利用にあたり、最も安定かつ手間がかからないという面では、以下の点が考えられます。

- **ディストリビューションの標準ストレージ・ドライバを使います** 。Docker をインストールする時、システム上の設定に応じてデフォルトのストレージ・ドライバを選択します。デフォルトのストレージ・ドライバの使用は、安定性に対する重要な要素になります。デフォルトのものを使わなければ、バグや微妙な差違に遭遇する可能性が増えるかもしれません。
- **CS Engine 互換表**[*1] **の詳細内容をご確認ください**。CS Engine とは商用サポート版の Docker Engine です。コード基盤はオープンソース版の Docker Engine と同じですが、ある範囲における設定をサポートしています。これらサポートされている設定の範囲では、最も安定かつ成熟したストレージ・ドライバを使います。これらの設定から外れれば、バグや微妙な差違に遭遇する可能性が増えるかもしれません。

経験と専門知識

ストレージ・ドライバの選択には、あなたと皆さんのチーム・組織で使ったことがあるものを選びます。例えば、RHEL[*2] や派生ディストリビューションを使っている場合は、既に LVM と Device Mapper の使用経験があるでしょう。その場合は、devicemapper ドライバの使用が望ましいでしょう。

Docker がサポートしているストレージ・ドライバの利用経験がないのであれば、どうしたら良いでしょうか。簡単に使える安定した Docker を使いたいのであれば、ディストリビューションが提供する Docker パッケージを使い、そこで使われているデフォルトのドライバの使用を検討すべきでしょう。

将来性の考慮

多くの方が OverlayFS こそが Docker ストレージ・ドライバの未来だと考えています。ですが、まだ成熟しておらず、安定性に関しては aufs や devicemapper のような成熟したドライバより劣るかもしれません。そのため、OverlayFS を注意して使用すべきであり、成熟したドライバを使うよりも多くのバグや差違に遭遇することが予想されます。

以下の図はストレージ・ドライバの一覧にしたものです。それぞれの良い点・悪い点に関する洞察をもたらすでしょう。

*1 https://www.docker.com/compatibility-maintenance

*2 Red Hat Enterprise Linux

5.4.3 AUFS ストレージ・ドライバの使用

AUFS（エーユーエフエス）は Docker に使われた初めてのストレージ・ドライバです。そのため、Docker の歴史で長く使われており、非常に安定し、多くの実際の開発に使われ、強力なコミュニティのサポートがあります。AUFS には複数の機能があります。これらは Docker の良い選択肢となるでしょう。次の機能を有効にします。

- コンテナ起動時間の高速化
- ストレージの効率的な利用
- メモリの効率的な利用

性能に拘わらず Docker で長い間使われてきていますが、いくつかのディストリビューションは AUFS をサポートしていません。たいていの場合、AUFS は Linux カーネルのメインライン（upstream）ではないためです。

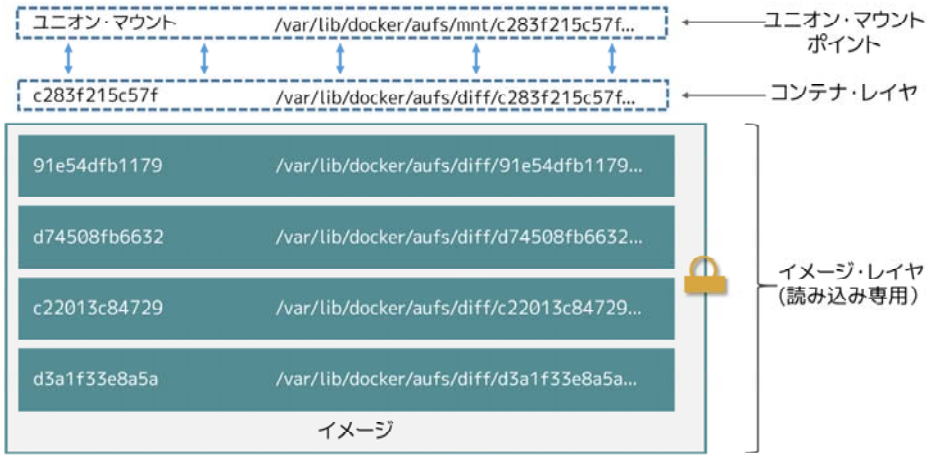
以下のセクションでは、AUFS 機能と Docker がどのように連携するか紹介します。

AUFS でイメージのレイヤ化と共有

AUFS とは**統合ファイルシステム（unification filesystem）**です。つまり、1 つの Linux ホスト上に複数のディレクトリが存在し、それぞれが互いに積み重なり、1 つに結合された状態に見えます。これらを実現するため、AUFS は**ユニオン・マウント（union mount）**を使います。

AUFS は複数のディレクトリの積み重ねであり、1 つのマウントポイントを通して、それらが統合されて見えます。全てのディレクトリが層（スタック）と結合したマウントポイントを形成しますので、全てが同一の Linux ホスト上に存在する必要があります。AUFS は各ディレクトリを、ブランチ（branch）という層の積み重ねとして参照します。

Docker 内部では、AUFS ユニオン・マウントがイメージのレイヤ化を行います。AUFS ストレージ・ドライバは、ユニオン・マウント・システムを使って Docker イメージを扱います。AUFS ブランチが Docker イメージ・レイヤに相当します。以下の図は `ubuntu:latest` イメージをベースとする Docker コンテナです。



この図は、Docker イメージ・レイヤと、Docker ホスト上の `/var/lib/docker` 以下にあるローカル・ストレージ領域との関係性を表しています。ユニオン・マウント・ポイントは、全てのレイヤを一体化して見えるようにします。Docker 1.10 からは、イメージ ID はデータが置かれるディレクトリ名と対応しなくなりました。

また、AUFS はコピー・オン・ライト技術（copy-on-write; CoW）もサポートしています。これは、全てのドライバがサポートしているものではありません。

AUFS でコンテナの読み書き

Docker は AUFS のコピー・オン・ライト技術をテコに、イメージ共有とディスク使用量の最小化をできるよう

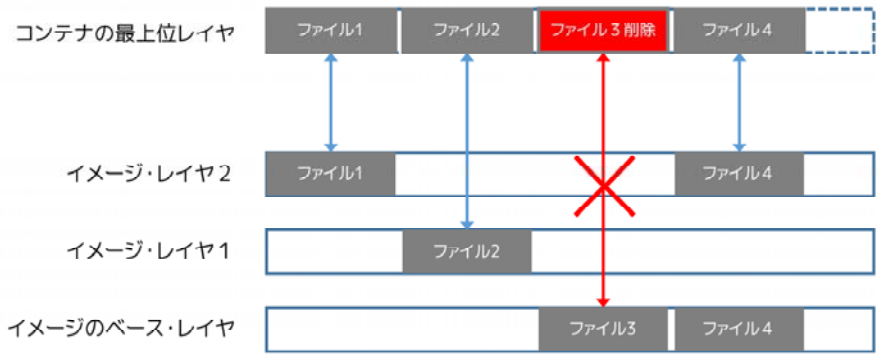
にします。AUFS はファイルレベルで動作します。つまり、AUFS のコピー・オン・ライト処理は、ファイル全体をコピーします。それがファイルの一部を変更する場合でもです。この処理はコンテナの性能に大きな影響を与えます。特に、コピーする対象のファイルが大きい場合は、配下のイメージ・レイヤが多くあるか、あるいは、コピー・オン・ライト処理により深いディレクトリ・ツリーを検索する必要なためです。

考えてみましょう。例えばコンテナで実行しているアプリケーションが、大きなキーバリュー・ストア（ファイル）に新しい値を追加したとします。これが初回であれば、コンテナの一番上の書き込み可能なレイヤに、まだ変更を加えるべきファイルが存在していません。そのため、コピー・オン・ライト処理は、下部のイメージからファイルを 上にコピー する必要があります。AUFS ストレージ・ドライバは、各イメージ・レイヤ上でファイルを探します。検索順番は、上から下にかけてです。ファイルが見つかれば、対象のファイルをコンテナの上にある書き込み可能なレイヤに コピー 開始処理（copy up）をします。そして、やっとファイルを開き、編集できるようになります。

小さなファイルに比べて、大きなファイルであれば明らかにコピー時間がかかります。そして、高いイメージ・レイヤにファイルがあるよりも、低いレイヤにある場合も時間がかかります。しかしながら、コピー作業が発生するのは、対象のコンテナ上では 1 度だけです。次にファイルの読み書き処理が発生しても、コンテナの一番上のレイヤにコピー済みのファイルがある場合は、ファイルを再度コピーしません。

AUFS ストレージ・ドライバでのファイル削除

AUFS ストレージ・ドライバでコンテナからファイルを削除したら、コンテナの一番上のレイヤに ホワイトアウト・ファイル（whiteout file） が置かれます。読み込み専用のイメージ・レイヤの下にあるファイルを、ホワイトアウト・ファイルが効果的に隠します。以下の単純化した図は、3 つのイメージ・レイヤのイメージに基づくコンテナを表しています。



Docker コンテナ
(AUFS ストレージ・ドライバのホワイトアウト・ファイルの説明用)

ファイル 3 はコンテナ上で削除されました。すると、AUFS ストレージ・ドライバは、コンテナの最上位レイヤにホワイトアウト・ファイルを置きます。このホワイトアウト・ファイルは、イメージの読み込み専用レイヤに存在するオリジナルのファイルを隠すことにより、コンテナ上から事実上 ファイル 3 が削除されたものとします。この処理はイメージの読み込み専用レイヤに対し何ら影響を与えません。

Docker で AUFS を使う設定

AUFS ストレージ・ドライバを使えるのは、AUFS がインストールされた Linux システム上でのみです。以下のコマンドを使い、システムが AUFS をサポートしているかどうか確認します。

```
$ grep aufs /proc/filesystems
nodev   aufs
```

この出力は、システムが AUFS をサポートしています。自分のシステムで AUFS をサポートしているのを確認

したら、Docker デーモンに対して AUFS を使う指示が必要です。これには docker daemon コマンドを使えます。

```
$ sudo docker daemon --storage-driver=aufs &
```

あるいは、Docker の設定ファイルを編集し、 DOCKER_OPTS 行に --storage-driver=aufs オプションを追加します。

```
# DOCKER_OPTS で、デーモン起動時のオプションを編集
DOCKER_OPTS="--storage-driver=aufs"
```

デーモンを起動したら、docker info コマンドでストレージ・ドライバを確認します。

```
$ sudo docker info
Containers: 1
Images: 4
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 6
 Dirperm1 Supported: false
Execution Driver: native-0.2
...出力を省略...
```

このような出力から、起動中の Docker デーモンが既存の ext4（イーエックスティーフォー） ファイルシステム上で AUFS ストレージ・ドライバを使っていることが分かります。

ローカルのストレージと AUFS

docker daemon を AUFS ドライバで実行したら、ドライバは Docker ホスト上のローカル・ストレージ領域である /var/lib/docker/aufs 内に、イメージとコンテナを保管します。

イメージ

イメージ・レイヤと各コンテナは、 /var/lib/docker/aufs/diff/<イメージ ID> ディレクトリ以下に保管されます。Docker 1.10 以降では、イメージ・レイヤ ID はディレクトリ名と一致しません。

/var/lib/docker/aufs/layers/ ディレクトリに含まれるのは、どのようにイメージ・レイヤを重ねるかというメタデータです。このディレクトリには、Docker ホスト上のイメージかコンテナごとに 1 つのファイルがあります（ファイル名はイメージのレイヤ ID と一致しません）。各ファイルの中にはイメージ・レイヤの名前があります。次の図は 1 つのイメージが 4 つのレイヤを持つのを示しています。

イメージ

以下のコマンドは、 /var/lib/docker/aufs/layers/ にあるメタデータ・ファイルを表示しています。ここで表示されるディレクトリの一覧は、ユニオン・マウントに積み重ねられている（スタックしている）ものです。ただし、覚えておかなくてはいけないのは、Docker 1.10 以上ではディレクトリ名とイメージ・レイヤ ID が一致しなくなりました。

```
$ cat /var/lib/docker/aufs/layers/91e54dfb11794fad694460162bf0cb0a4fa710cfa3f60979c177d920813e267c
d74508fb6632491cea586a1fd7d748dfc5274cd6fdfedee309ecdcbc2bf5cb82
c22013c8472965aa5b62559f2b540cd440716ef149756e7b958a1b2aba421e87
d3a1f33e8a5a513092f01bb7eb1c2abf4d711e5105390a3fe1ae2248cfde1391
```

イメージのベース・レイヤは下層にイメージ・レイヤを持ちませんので、対象となるファイルの内容は空っぽです。

コンテナ

実行中のコンテナは /var/lib/docker/aufs/mnt/<コンテナ ID> 配下にマウントされます。これが AUFS ユニオン・マウント・ポイントであり、コンテナと下層のイメージ・レイヤが 1 つに統合されて公開されている場所です。コンテナが実行されていなければ、これらのディレクトリは存在しますが、内容は空っぽです。なぜなら、コンテナが実行する時のみマウントするための場所だからです。Docker 1.10 以上では、コンテナ ID はディレクトリ名 /var/lib/docker/aufs/mnt/<コンテナ ID> と対応しません。

コンテナのメタデータやコンテナの実行に関する様々な設定ファイルは、 /var/lib/docker/containers/<コンテナ ID> に保管されます。ディレクトリ内に存在するファイルはシステム上の全コンテナに関するものであり、停止されたものも含みます。しかしながら、コンテナを実行したら、コンテナのログファイルもこのディレクトリに保存されます。

コンテナの**薄い書き込み可能なレイヤ**（**thin writable layer**）は /var/lib/docker/aufs/diff/<コンテナ ID> に保存されます。Docker 1.10 以上では、コンテナ ID はディレクトリ名と対応しません。しかしながら、コンテナの薄い書き込み可能なレイヤは、まだこの配下に存在し続けています。このディレクトリは AUFS によってコンテナの最上位の書き込みレイヤとして積み重ねられるものであり、コンテナに対する全ての変更が保管されます。コンテナが停止しても、このディレクトリは存在し続けます。つまり、コンテナを再起動しても、その変更内容は失われません。コンテナが削除された時のみ、このディレクトリは削除されます。

AUFS と Docker の性能

既に言及している性能面について、まとめます。

- AUFS ストレージ・ドライバは PaaS とコンテナの密度が重要な類似事例にとって、良い選択肢です。これは複数の実行中のコンテナ間で、AUFS が効率的にイメージを共有するためです。それにより、コンテナ

の起動時間を早くし、ディスク使用量を最小化します。

- AUFS がイメージ・レイヤとコンテナ間でどのように共有するのか、その根底にある仕組みは、システム・ページ・キャッシュを非常に効率的に使います。
- AUFS ストレージ・ドライバはコンテナに対する書き込み性能に対し、著しい待ち時間をもたらし得ます。これはコンテナに何らかのファイルを書き込もうとしたら、ファイルをコンテナ最上位の書き込み可能レイヤに対してコピーする必要があるためです。ファイルが多くのイメージ・レイヤに存在する場合や、ファイル自身が大きい場合には、待ち時間が増え、悪化するでしょう。

最後に 1 つだけ。データ・ボリュームは最高かつ最も予想可能な性能をもたらします。これはデータ・ボリュームがストレージ・ドライバを迂回するためであり、シン・プロビジョニングやコピー・オン・ライトによるオーバーヘッドの影響を受けないためです。そのため、重い書き込み処理を行いたい場合には、データ・ボリュームの使用が適している場合もあるでしょう。

5.4.4 Btrfs ストレージ・ドライバの使用

Btrfs（ビーツリー・エフエス）は次世代のコピー・オン・ライト対応ファイルシステムです。多くの高度なストレージ技術をサポートしており、Docker に適しています。Btrfs は Linux カーネルの主流に含まれており、オンディスク・フォーマット（on-disk format）[*1] は安定していると考えられています。しかしながら、その機能の大部分はまだ開発中です。そのため、ユーザとしては Btrfs とは動きの速い「モノ」と考えるべきでしょう。

Docker の **btrfs ストレージ・ドライバ**は、イメージとコンテナを管理するために、多くの Btrfs 機能を活用します。機能とは、シン・プロビジョニング、コピー・オン・ライト、スナップショットに関するものです。

このセクションでは、Docker の Btrfs ストレージ・ドライバを `btrfs` と表記します。また、Btrfs ファイルシステム全体を Btrfs と表記します。

商用サポート版 Docker Engine（CS-Engine）は、現時点では btrfs ストレージ・ドライバをサポートしていません。

Btrfs の未来

Btrfs は将来の Linux ファイルシステムとして、長く賞賛されています。Linux カーネルの主流として完全にサポートされ、安定したオン・ディスク・フォーマットと、安定性に焦点をあてた活発な開発が、より現実性を高めています。

Linux プラットフォームで Docker を使う限り、多くの皆さんには `devicemapper` ストレージ・ドライバを長期的に置き換えるものとして、`btrfs` ストレージドライバがあるように見えてしまうでしょう。しかし、これを書いている時点では、 `devicemapper` ストレージ・ドライバの方が安全で、より安定しており、より**プロダクションに対応している**と考えるべきです。`btrfs` ドライバを理解し、Btrfs の経験がある場合のみ、プロダクションの開発に `btrfs` ドライバを検討すべきでしょう。

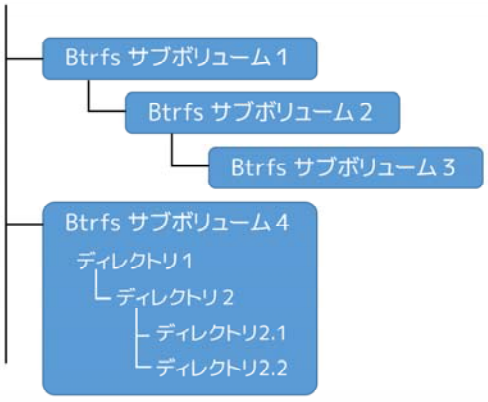
Btrfs でイメージのレイヤ化と共有

ディスク上のイメージの構成物とコンテナ・レイヤを管理するため、Docker は Btrfs の**サブボリューム**と**スナップショット**を活用します。Btrfs サブボリュームの見た目は、通常の Unix ファイルシステムと同じです。各々が内部にディレクトリ構造を持つのは、幅広い Unix ファイルシステムで見られるものです。

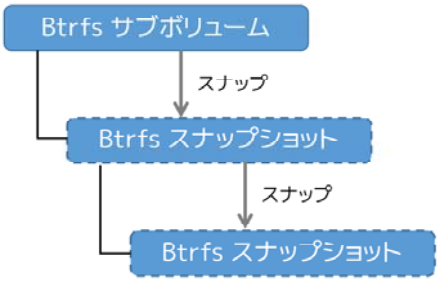
サブボリュームはコピー・オン・ライトにネイティブに対応しており、必要があれば基礎をなすストレージ・プールから領域を割り当てます。また、領域はネスト（入れ子）にすることができ、スナップ化できます。次の図は

*1 訳者注：Btrfs のデータ管理技術です。

４つのボリュームを表します。「サブボリューム４」は内部にディレクトリ・ツリーを持っているのに対し、「サブボリューム２」と「サブボリューム３」はネストされたものです。



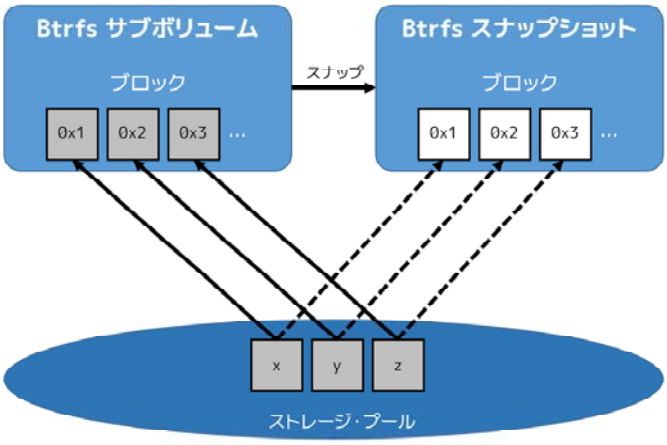
スナップショットとは、ある時点におけるサブボリューム全体の読み書きをコピーします。既存のディレクトリの下にサブボリュームを作成します。下図のように、スナップショットのスナップショットも作成できます。



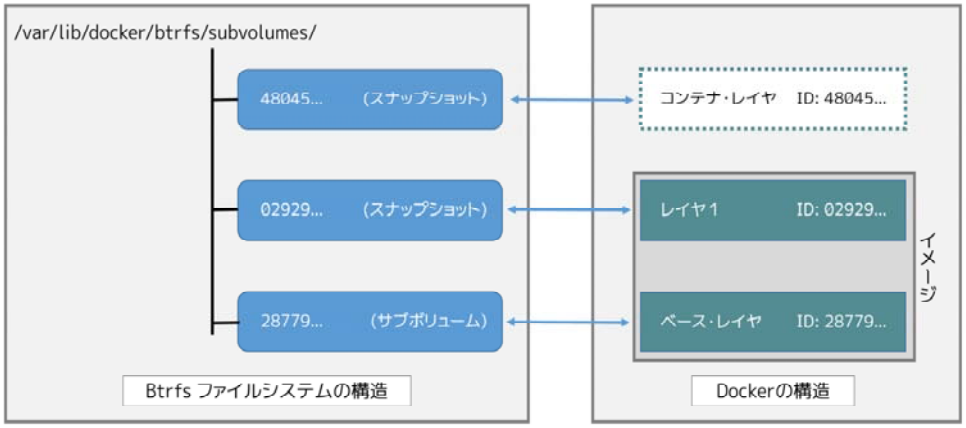
サブボリュームとスナップショットの必要に応じ、Btrfs は基盤をなすストレージ・プールから領域を割り当てます。割り当ての単位は chunk（チャンク） と chunks（チャンクス） から参照され、通常は 1GB 以下の大きさです。

スナップショットは Btrfs ファイルシステム上の優秀な機能です。つまり、通常のサブボリュームと同じ見た目と、感触、操作が可能です。ネイティブなコピー・オン・ライトの設計のおかげで、Btrfs ファイルシステム内のディレクトリにサブボリュームを構築する時、この技術が必要になります。つまり、Btrfs スナップショットは性能のオーバーヘッドが少ない、あるいは無いため、効率的に領域を使います。次の図はスナップショットが同じデータを共有しているサブボリュームです。

Docker の btrfs ストレージ・ドライバは、各イメージ・レイヤとコンテナを、自身の Btrfs サブボリュームかス



ナップショットに保管します。イメージのベース・レイヤはサブボリュームとして保管します。それに対して子イメージ・レイヤとコンテナは、スナップショットに保管します。これを説明したのが次の図です。

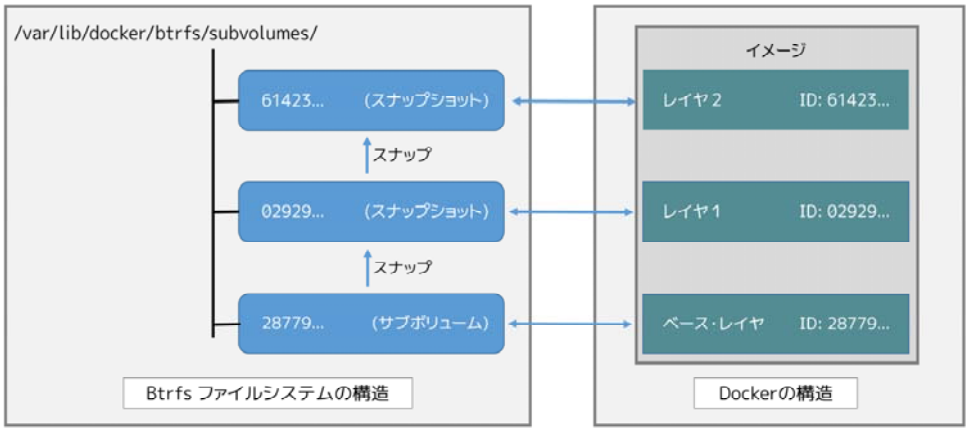


Docker ホストが btrfs ドライバを使い、イメージとコンテナの作成という高レベルの処理手順は、次のように行います。

1. イメージのベース・レイヤは `/var/lib/docker/btrfs/subvolumes` 以下の Btrfs サブボリュームに保管します。

2. 以降のイメージ・レイヤは、親レイヤのサブボリュームの Btrfs スナップショットとして保存されるか、(単体の)スナップショットになります。

以下の図は 3 つのイメージ・レイヤを表しています。ベース・レイヤはサブボリュームです。レイヤ 1 はベース・レイヤに対するサブボリュームのスナップショットです。レイヤ 2 はレイヤ 1 のスナップショットです。



Docker 1.10 からは、イメージ・レイヤ ID は `/var/lib/docker` 以下のディレクトリ名と一致しません。

ディスク構造上のイメージとコンテナ

イメージ・レイヤとコンテナは、Docker ホスト上のファイルシステム `/var/lib/docker/btrfs/subvolumes/` にあります。しかしながら、以前とは異なり、ディレクトリ名はイメージ ID の名前を表しません。コンテナ用のディレクトリは、コンテナが停止した状態でも表示されます。ストレージ・ドライバがデフォルトでマウントするのは、 `/var/lib/docker/subvolumes` のサブボリュームをトップレベルとする場所です。その他全てのサブボリュームとボリューム名は、Btrfs ファイルシステムのオブジェクトとして個々にマウントするのではなく、この下に存在しています。

Btrfs はファイルシステム・レベルで動作するものであり、ブロック・レベルではありません。各イメージとコンテナのレイヤは、通常の Unix コマンドを使って参照できます。次の例は、イメージの最上位レイヤに対して `ls -l` コマンドを実行した結果を省略したものです。

```
$ ls -l
/var/lib/docker/btrfs/subvolumes/0a17decee4139b0de68478f149cc16346f5e711c5ae3bb969895f22dd6723751/
total 0
drwxr-xr-x 1 root root 1372 Oct  9 08:39 bin
drwxr-xr-x 1 root root    0 Apr 10  2014 boot
drwxr-xr-x 1 root root  882 Oct  9 08:38 dev
drwxr-xr-x 1 root root 2040 Oct 12 17:27 etc
drwxr-xr-x 1 root root    0 Apr 10  2014 home
...表示結果を省略...
```

Btrfs でコンテナを読み書き

コンテナはイメージ領域を効率的に扱うスナップショットです。スナップショットの中のメタデータが示す実際のデータ・ブロックは、ストレージ・プールの中にあります。これはサブボリュームの扱いと同じです。そのため、スナップショットの読み込み性能は、サブボリュームの読み込み性能と本質的に同じです。その結果、Btrfs ドライバ使用による性能のオーバーヘッドはありません。

新しいファイルをコンテナに書き込む時、コンテナのスナップショットに新しいデータブロックを割り当てる処理が、必要に応じて発生します。それから、ファイルを新しい領域に書き込みます。必要に応じて書き込む処理は Btrfs によってネイティブに書き込まれ、新しいデータをサブボリュームに書き込むのと同じです。その結果、コンテナのスナップショットに新しいファイルを書き込む処理は、ネイティブな Btrfs の速度になります。

コンテナ内にある既存のファイルを更新したら、コピー・オン・ライト処理（技術的には、書き込みのための転送、という意味です）が発生します。ドライバはオリジナルのデータをそのままに、スナップショットに新しい領域を割り当てます。更新されたデータは新しい領域に書き込みます。それから、ドライバはファイルシステムのメタデータを更新し、スナップショットが新しいデータを示すようにします。元々あったデータはサブボリュームとスナップショットのための更なるツリーの活用場所として維持されます。この動作はコピー・オン・ライトのファイルシステムがネイティブな Btrfs 向けであり、非常に小さなオーバーヘットとなります。

Btrfs を使う場合、大量の小さなファイルの書き込みと更新は、パフォーマンスの低下を招きます。詳細は後ほど扱います。

Docker で Btrfs を設定

`btrfs` ストレージ・ドライバは、Docker ホストで Btrfs ファイルシステムとしてマウントしている `/var/lib/docker` のみ処理します。以下の手順で、Ubuntu 14.04 LTS 上で Btrfs を設定する方法を紹介します。

動作条件

既に Docker ホスト上で Docker デーモンを使っている場合は、イメージをどこかに保存する必要があります。そのため、処理を進める前に、それらのイメージを Docker Hub やプライベート Docker Trusted Registry に送信しておきます。

まず Docker デーモンを停止します。そして `/dev/xvdb` に予備のブロック・デバイスがあることを確認します。このデバイスは個々の環境によって違うかもしれませんが、処理にあたっては各環境によって違う場合があります。

またこの手順では、カーネルが適切な Btrfs モジュールを読み込まれているものと想定しています。これらを確認したら、以下のコマンドを実行します。

```
$ cat /proc/filesystems | grep btrfs`
```

Ubuntu 14.04 LTS で Btrfs を設定

システムが動作条件を満たしていると仮定し、次の手順を進めます。

1. 「btrfs-tools」パッケージをインストールします。

```
$ sudo apt-get install btrfs-tools
Reading package lists... Done
Building dependency tree
<出力を省略>
```

2. Btrfs ストレージ・プールを作成します。

Btrfs ストレージ・プールは mkfs.btrfs コマンドで作成します。複数のデバイスにわたるプールを作成するには、それぞれのデバイスで mkfs.btrfs コマンドを実行します。ここでは、作成したプールは単一デバイス上の /dev/xvdb と想定しています。

```
$ sudo mkfs.btrfs -f /dev/xvdb
WARNING! - Btrfs v3.12 IS EXPERIMENTAL
WARNING! - see http://btrfs.wiki.kernel.org before using


Turning ON incompat feature 'extref': increased hardlink limit per file to 65536
fs created label (null) on /dev/xvdb
    nodesize 16384 leafsize 16384 sectorsize 4096 size 4.00GiB
Btrfs v3.12
```

/dev/xvdb には、各システム上の適切なデバイスを割り当ててください。

【警告】Btrfs は実験的な実装なのでご注意ください。先ほど説明した通り、Btrfs の利用経験がなければ、現時点ではプロダクションへのデプロイには推奨されていません。

3. Docker ホスト上に、ローカル・ストレージ領域が /var/lib/docker になければ、ディレクトリを作成します。

```
$ sudo mkdir /var/lib/docker
```

4. システムのブート時に、毎回自動的に Btrfs ファイルシステムをマウントするよう設定します。

a. Btrfs ファイルシステムの UUID を取得します。

```
$ sudo blkid /dev/xvdb
/dev/xvdb: UUID="a0ed851e-158b-4120-8416-c9b072c8cf47"
UUID_SUB="c3927a64-4454-4eef-95c2-a7d44ac0cf27" TYPE="btrfs"
```

b. /etc/fstab エントリに、システムのブート時に毎回自動的に /var/lib/docker をマウントする記述を追加します。

```
/dev/xvdb /var/lib/docker btrfs defaults 0 0
UUID="a0ed851e-158b-4120-8416-c9b072c8cf47" /var/lib/docker btrfs defaults 0 0
```

5. 新しいファイルシステムをマウントし、操作可能か確認します。

```
$ sudo mount -a
$ mount
/dev/xvda1 on / type ext4 (rw,discard)
<出力を省略>
/dev/xvdb on /var/lib/docker type btrfs (rw)
```

最後の行の出力から、 /dev/xvdb は Btrfs として /var/lib/docker をマウントしているのが分かります。

これで Btrfs ファイルシステムが /var/lib/docker をマウントしたので、デーモンは btrfs ストレージ・ドライバを自動的に読み込めるようにします。

1. Docker デーモンを起動します。

```
$ sudo service docker start
docker start/running, process 2315
```

この Docker デーモンの起動手順は、使用している Linux ディストリビューションによっては異なる場合があります。

btrfs ストレージ・ドライバを使って Docker デーモンを起動するには、docker daemon コマンドで --storage-driver=btrfs フラグを渡すか、Docker 設定ファイルの DOCKER_OPT 行でフラグを追加します。

2. docker info コマンドでストレージ・ドライバを確認します。

```
$ sudo docker info
Containers: 0
Images: 0
Storage Driver: btrfs
[...]
```

これで、Docker ホストは btrfs ストレージ・ドライバを使う新しい設定で動いています。

Btrfs と Docker の性能

btrfs ストレージ・ドライバ配下では、Docker の性能に影響を与える様々な要素があります。

- **ページ・キャッシュ**：Btrfs はページ・キャッシュ共有をサポートしていません。つまり、 n 個のコンテナがキャッシュするために、 n 個のコピーを必要とします。そのため、btrfs ドライバは、PaaS や、その他にも密度を求められる用途には適していません。
- **小さな書き込み**：コンテナに対する小さな書き込み（Docker ホストが多くのコンテナを起動・停止している場合）が大量であれば、Btrfs の塊（chunk）を粗く消費します。これにより、停止するまで Docker ホスト上で多くの容量を消費します。これが現時点の Btrfs バージョンにおける主な障害です。

もし btrfs ストレージ・ドライバを使うのであれば、btrfs filesystem show コマンドで Btrfs ファイルシステム上の空き容量を頻繁に監視してください。 df のような通常の Unix コマンドを信頼しないでください。つまり、常に Btrfs のネイティブ・コマンドを使ってください。

- **シーケンシャルな書き込み**：Btrfs はジャーナリングの技術を使ってデータをディスクに書き込みます。この影響により、シーケンシャル（連続した）書き込みでは、性能が半減します。

- **断片化（fragmentation）**：断片化とは、Btrfs のようなフィルシステム上でコピー・オン・ライトを行うと生じる自然な副産物です。たくさんの小さなランダムな書き込みが、断片化を引き起こします。SSD メディアを使う Docker ホスト上では CPU のスパイク（突発的な利用）が顕著ですし、回転メディア（HDD）で Docker ホストを動かす場合も、メディアをむち打つものです。いずれの場合も、パフォーマンス低下の影響を招きます。

Btrfs の最近のバージョンは、autodefrag をマウント用のオプションに指定できます。このモードによって、断片化せずにランダムに書き込みをします。ただし、Docker ホスト上でこのオプションを有効化する前に、自分自身で性能評価をすべきです。いくつかのテストは Docker ホスト上に多数の小さなファイルを作成しますので、良くない性能に影響与える場合があります（あるいは、システムで沢山のコンテナを停止・起動した場合も）。

- **SSD（ソリッド・ステート・ドライブ）**：Btrfs は SSD メディアをネイティブに最適化します。最適化を有効化するには、マウントオプションで `-o ssh` を指定します。これらの最適化には、SSD はメディアを使わずシーク最適化が不要なため、これら最適化により SSD の性能を拡張します。

また、Btrfs は TRIM/Discard プリミティブもサポートします。しかし、 `-o discard` マウント・オプションでマウントすると、性能の問題を引き起こします。そのため、このオプションを使う前に、自分自身で性能評価をするのを推奨します。

- **データ・ボリュームの使用**：データ・ボリュームは最上かつ最も予測可能な性能を提供します。これは、ストレージ・ドライバを迂回し、シン・プロビジョニングやコピー・オン・ライト処理を行わないためです。そのため、データ・ボリューム上で重たい書き込みを行うのに適しています。

5.4.5 Device Mapper ストレージ・ドライバの使用

Device Mapper は、Linux 上で多くの高度なボリューム管理技術を支えるカーネル・ベースのフレームワークです。Docker の **devicemapper ストレージ・ドライバ**は、シン・プロビジョニングとスナップショット機能のために、イメージとコンテナ管理にこのフレームワークを活用します。このセクションでは、Device Mapper ストレージ・ドライバを `devicemapper` と表記します。カーネルのフレームワークを Device Mapper として表記します。

`devicemapper` ストレージ・ドライバを使うには、RHEL か CentOS Linux 上で商用サポート版 Docker Engine (CS-Engine) を実行する必要があります。

AUFS の代替

当初の Docker は、Ubuntu と Debian Linux 上で AUFS をストレージのバックエンドに使っていました。Docker が有名になるにつれ、多くの会社が Red Hat Enterprise Linux 上で使いたいと考え始めました。残念ながら、AUFS は Linux カーネル上流のメインラインではないため、RHEL は AUFS を扱いませんでした。

この状況を変えるべく、Red Hat 社の開発者らが AUFS をカーネルのメインラインに入れられるよう取り組みました。しかしながら、新しいストレージ・バックエンドを開発する方が良い考えであると決断したのです。更に、ストレージのバックエンドには、既に存在していた Device Mapper 技術を基盤としました。

Red Hat 社は Docker 社と協同で新しいドライバの開発に取り組みました。この協調の結果、ストレージ・バックエンドの取り付け・取り外しが可能な（pluggable）Docker エンジンを再設計しました。そして、`devicemapper` は Docker がサポートする 2 つめのストレージ・ドライバとなったのです。

Device Mapper は Linux カーネルのバージョン 2.6.9 以降、メインラインに組み込まれました。これは、RHEL ファミリーの Linux ディストリビューションの中心部です。つまり、`devicemapper` ストレージ・ドライバは安定したコードを基盤としており、現実世界における多くのプロダクションへのデプロイをもたらします。また、強力

なコミュニティのサポートも得られます。

イメージのレイヤ化と共有

devicemapper ドライバは、全てのイメージとコンテナを自身の仮想デバイスに保管します。これらのデバイスとは、シン・プロビジョニングされ、コピー・オン・ライト可能であり、スナップショットのデバイスです。Device Mapper 技術はファイル・レベルというよりも、ブロック・レベルで動作します。つまり、 devicemapper ストレージ・ドライバの**シン・プロビジョニング**（**thin-provisioning**）と**コピー・オン・ライト**（**copy-on-write**）処理は、ファイルではなくブロックに対して行います。

また、スナップショットは シン・デバイス（thin device） や仮想デバイス（virtual device）としても参照されます。つまり、 devicemapper ストレージ・ドライバにおいては、どれも同じものを意味します。

devicemapper でイメージを作る高度な手順は、以下の通りです。

1. devicemapper ストレージ・ドライバは**シン・プール**（**thin pool**）を作成します。

ブロック・デバイスかループ用にマウントした**薄いファイル**（**sparse files**）上に、このプールを作成します。

2. 次に**ベース・デバイス**（**base device**）を作成します。

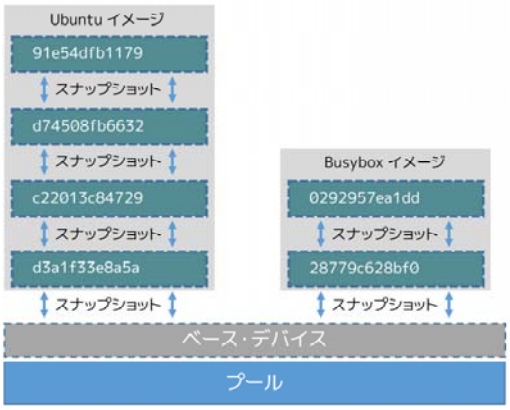
ベース・デバイスとは、ファイルシステムのシン・デバイス（thin device）です。どのファイルシステムが使われているかを調べるには、 docker info コマンドを実行し、Backing filesystem（バッキングファイルシステム） 値を確認します。

3. それぞれの新しいイメージ（とイメージ・レイヤ）は、このベース・デバイスのスナップショットです。

これらがシン・プロビジョニングされたコピー・オン・ライトなスナップショットです。つまり、これらは初期状態では空っぽですが、データが書き込まれる時だけ容量を使います。

devicemapper では、ここで作成されたイメージのスナップショットが、コンテナ・レイヤになります。イメージと同様に、コンテナのスナップショットも、シン・プロビジョニングされたコピー・オン・ライトなスナップショットです。コンテナのスナップショットに、コンテナ上での全ての変更が保管されます。devicemapper は、コンテナに対してデータを書き込む時とき、このプールから必要に応じて領域を割り当てます。

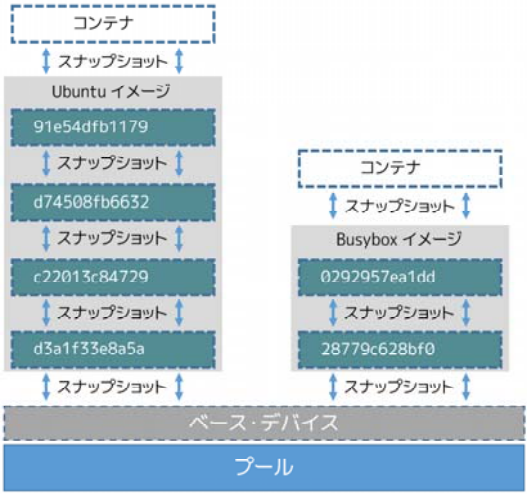
以下のハイレベルな図は、ベース・デバイスのシン・プールと２つのイメージを表します。



細かく図を見ていきますと、スナップショットは全体的に下向きなのが分かるでしょう。各イメージ・レイヤは下にあるレイヤのスナップショットです。各イメージの最も下にあるレイヤは、プール上に存在するベース・デバ

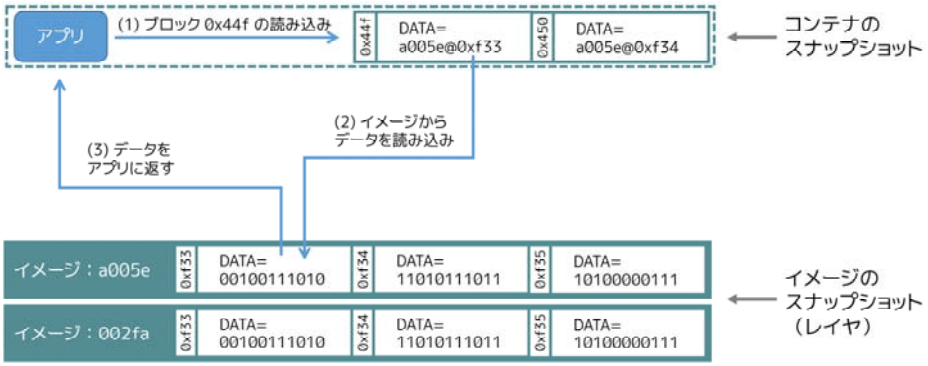
イスのスナップショットです。このベース・デバイスとは Device Mapper の**アーティファクト（artifact；成果物の意味）**であり、Docker イメージ・レイヤではありません。

コンテナとは、ここから作成したイメージのスナップショットです。下図は２つのコンテナです。一方が Ubuntu イメージをベースにし、もう一方は Busybox イメージをベースにしています。



devicemapper からの読み込み

devicemapper ストレージ・ドライバが、どのように読み書きしているか見ていきましょう。下図は、サンプル・コンテナが単一のブロック（0x44f）を読み込むという、ハイレベルな手順です。



1. アプリケーションがコンテナ内のブロック 0x44f に対して読み込みを要求します。

コンテナは、イメージの**薄い（thin）**スナップショットであり、データを持っていません。その代わりに、下層のイメージ層（スタック）にあるイメージのスナップショット上の、どこにデータが保管されているかを示すポインタ（PTR）を持っています。

2. ストレージ・ドライバは、スナップショットのブロック 0xf33 と関連するイメージ・レイヤ a005... のポインタを探します。

3. devicemapper はブロック 0xf33 の内容を、イメージのスナップショットからコンテナのメモリ上にコピーします。

4. ストレージ・ドライバはアプリケーションがリクエストしたデータを返します。

書き込み例

devicemapper ドライバで新しいデータをコンテナに書き込むには、**オンデマンドの割り当て（allocate-on-demand）**を行います。コピー・オン・ライト処理により、既存のデータを更新します。Device Mapper はブロック・ベースの技術のため、これらの処理をブロック・レベルで行います。

例えば、コンテナ内の大きなファイルに小さな変更を加える時、devicemapper ストレージ・ドライバはファイル全体コピーをコピーしません。コピーするのは、変更するブロックのみです。各ブロックは 64KB です。

新しいデータの書き込み

コンテナに 56KB の新しいデータを書き込みます。

1. アプリケーションはコンテナに 56KB の新しいデータの書き込みを要求します。

2. オンデマンドの割り当て処理により、コンテナのスナップショットに対して、新しい 64KB のブロックが 1 つ割り当てられます。

書き込み対象が 64KB よりも大きければ、複数の新しいブロックがコンテナに対して割り当てられます。

3. 新しく割り当てられたブロックにデータを書き込みます。

既存のデータを上書き

既存のデータに対して初めて変更を加える場合、

1. アプリケーションはコンテナ上にあるデータの変更を要求します。

2. 更新が必要なブロックに対して、コピー・オン・ライト処理が行われます。

3. 処理によって新しい空のブロックがコンテナのスナップショットに割り当てられ、そのブロックにデータがコピーされます。

4. 新しく割り当てられたブロックの中に、変更したデータを書き込みます。

コンテナ内のアプリケーションは、必要に応じた割り当てやコピー・オン・ライト処理を意識しません。しかしながら、アプリケーションの読み書き処理において、待ち時間を増やすでしょう。

Device Mapper を Docker で使う設定

複数のディストリビューションにおいて、devicemapper は標準の Docker ストレージ・ドライバです。ディストリビューションには RHEL や派生したものが含まれます。現時点では、以下のディストリビューションがドライバをサポートしています。

- RHEL/CentOS/Fedora
- Ubuntu 12.04
- Ubuntu 14.04
- Debian

Docker ホストは devicemapper ストレージ・ドライバを、デフォルトでは **loop-lvm（ループエルブイエム） モード**で設定します。このモードは、イメージとコンテナのスナップショットが使うシン・プール（thin pool）を構築するために、**スパース・ファイル（sparse file；まばらなファイル）**を使う指定です。このモードは、設定に変更を加えることなく、革新的な動きをするように設計されています。しかしながら、プロダクションへのデプロイでは、loop-lvm モードの下で実行すべきではありません。

どのようなモードで動作しているか確認するには docker info コマンドを使います。

```
$ sudo docker info
Containers: 0
Images: 0
Storage Driver: devicemapper
 Pool Name: docker-202:2-25220302-pool
 Pool Blocksize: 65.54 kB
 Backing Filesystem: xfs
 ...
 Data loop file: /var/lib/docker/devicemapper/devicemapper/data
 Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata
 Library Version: 1.02.93-RHEL7 (2015-01-28)
 ...
```

この実行結果から、Docker ホストは devicemapper ストレージ・ドライバの処理に loop-lvm モードを使っているのが分かります。実際には、**データ・ループ・ファイル（data loop file）**と**メタデータ・ループ・ファイル（Metadata loop file）**のファイルが /var/lib/docker/devicemapper/devicemapper 配下にあるのを意味します。これらがループバックにマウントされているパース・ファイルです。

プロダクション用に direct-lvm モードを設定

プロダクションへのデプロイに適した設定は **direct lvm（ダイレクトエルブイエム） モード**です。このモードはシン・プールの作成にブロック・デバイスを使います。以下の手順は、Docker ホストが devicemapper ストレージ・ドライバを direct-lvm 設定で使えるようにします。

既に Docker ホスト上で Docker デーモンを使っている場合は、イメージをどこかに保存する必要があります。そのため、処理を進める前に、それらのイメージを Docker Hub やプライベート Docker Trusted Registry に送信しておきます。

以下の手順は 90GB のデータ・ボリュームと 4GB のメタデータ・ボリュームを作成し、ストレージ・プールの基礎として使います。ここでは別のブロック・デバイス /dev/sdd を持っており、処理するための十分な空き容量があると想定しています。デバイスの識別子とボリューム・サイズは皆さんの環境とは異なるかもしれません。手順を進める時は、自分の環境にあわせて適切に置き換えてください。また、手順は Docker デーモンが停止した状態から始めるのを想定しています。

1. 設定対象の Docker ホストにログインします。

2. Engine のデーモンが実行中であれば、停止します。

3. LVM（論理ボリューム・マネジメント）のバージョン 2 をインストールします。

```
$ yum install lvm2
```

4. 物理ボリュームにブロック・デバイス /dev/sdd を作成します。

```
$ pvcreate /dev/sdd
```

5. docker ボリューム・グループを作成します。

```
$ vgcreate docker /dev/sdd
```

6. thinpool という名前のシン・プール（thin pool）を作成します。

この例では、docker ボリューム・グループの論理データ（data logical）は 95% の大きさとします。残りの容量は、データもしくはメタデータによって空き容量が少なくなった時の一時的な退避用に使います。

```
$ lvcreate --wipesignatures y -n thinpool docker -l 95%VG
$ lvcreate --wipesignatures y -n thinpoolmeta docker -l 1%VG
```

7. プールをシン・プールに変換します。

```
$ lvconvert -y --zero n -c 512K --thinpool docker/thinpool --poolmetadata docker/thinpoolmeta
```

8. lvm プロフィールを経由してシン・プールを自動拡張するよう設定します。

```
$ vi /etc/lvm/profile/docker-thinpool.profile
```

9. thin_pool_autoextend_threshold 値を指定します。

ここで指定する値は、先ほどの lvm 領域がどの程度まで到達したら、領域をどこまで自動拡張するかをパーセントで指定します（100 = 無効化です）。

```
thin_pool_autoextend_threshold = 80
```

10. シン・プールの自動拡張が発生するタイミングを指定します。

シン・プールの領域を増やす空き容量のタイミングをパーセントで指定します（100 = 無効化です）。

```
thin_pool_autoextend_percent = 20
```

11. 確認をします。docker-thinpool.profile は次のように表示されます。

/etc/lvm/profile/docker-thinpool.profile ファイルの例：

```
activation {
    thin_pool_autoextend_threshold=80
    thin_pool_autoextend_percent=20
}
```

12. 新しい lvm プロフィールを適用します。

```
$ lvchange --metadataprofile docker-thinpool docker/thinpool
```

13. lv（論理ボリューム）をモニタしているのを確認します。

```
$ lvs -o+seg_monitor
```

14. Docker Engine を起動していた場合は、グラフ・ドライバを直接削除します。

Docker インストール時のイメージとコンテナからグラフ・ドライバを削除します。

```
$ rm -rf /var/lib/docker/*
```

15. Engine デーモンが devicemapper オプションを使うように設定します。

設定には 2 つの方法があります。デーモンの起動時にオプションを指定するには、次のようにします。

```
--storage-driver=devicemapper --storage-opt=dm.thinpooldev=/dev/mapper/docker-thinpool --storage-opt
dm.use_deferred_removal=true
```

あるいは daemon.json 設定ファイルで起動時に指定もできます。例：

```
{
        "storage-driver": "devicemapper",
        "storage-opts": [
                "dm.thinpooldev=/dev/mapper/docker-thinpool",
                "dm.use_deferred_removal=true"
        ]
}
```

16. Docker Engine デーモンを起動します。

```
$ systemctl start docker
```

Docker Engine デーモンを起動したら、シン・プールとボリューム・グループの空き容量を確認します。ボリューム・グループは自動拡張しますので、容量を使い尽くす可能性があります。論理ボリュームを監視するには、オプションを指定せず lvs を使うか、lvs -a でデータとメタデータの大きさを確認します。ボリューム・グループの空き容量を確認するには vgs コマンドを使います。

先ほど設定したシン・プールの閾値を越えたかどうかを確認するには、次のようにログを表示します。

```
ジャーナルシーティーエル
journalctl -fu dm-event.service
```

シン・プールで問題を繰り返す場合は、dm.min_free_spaces オプションで Engine の挙動を調整できます。この値は最小値に近づいた時、警告を出して操作させなくします。詳しい情報は docker daemon リファレンスにあるストレージ・ドライバのオプションをご覧ください。

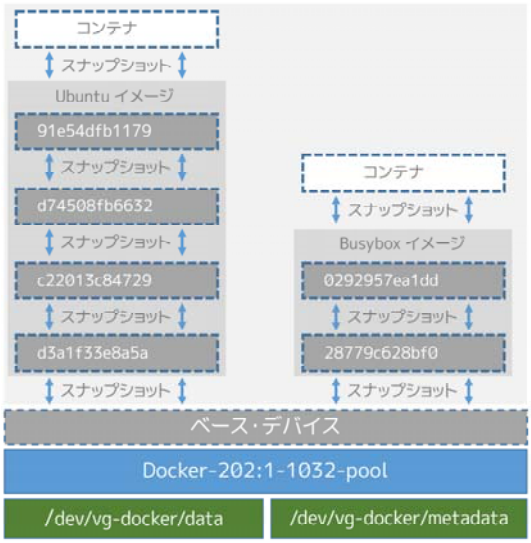
ホスト上の devicemapper 構造の例

エルエスブルク
lsblk コマンドを使えば、先ほど作成したデバイス・ファイルと、その上に devicemapper ストレージ・ドライ

バによって作られた pool（プール）を確認できます。

```
$ sudo lsblk
NAME                         MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
xvda                         202:0    0    8G  0 disk
└─xvda1                        202:1    0    8G  0 part /
xvdf                         202:80   0   10G  0 disk
├─vg--docker-data              253:0    0   90G  0 lvm
│ └─docker-202:1-1032-pool 253:2    0   10G  0 dm
└─vg--docker-metadata          253:1    0    4G  0 lvm
  └─docker-202:1-1032-pool 253:2    0   10G  0 dm
```

下図は、先ほどの例で扱ったイメージの更新を、 lsblk コマンドの詳細とあわせて表しています。



この図では、プールは Docker-202:1-1032-pool と名付けられ、先ほど作成した data と metadata デバイスにわたっています。この devicemapper のプール名は、次のような形式です。

```
Docker-MAJ:MIN-INO-pool
```

MAJ 、 NIN 、 INO は、デバイスのメジャー番号、マイナー番号、i ノード番号です。

Device Mapper はブロック・レベルで処理を行うため、イメージ・レイヤとコンテナ間の差分を見るのは、少し大変です。しかしながら、２つの鍵となるディレクトリがあります。 /var/lib/docker/devicemapper/mnt ディレクトリには、イメージとコンテナのマウント・ポイントがあります。 /var/lib/docker/devicemapper/metadata ディレクトリには、それぞれのイメージとコンテナのスナップショットを格納する１つのファイルがあります。このファイルには、各スナップショットのメタデータが JSON 形式で含まれています。

Device Mapper と Docker の性能

オンデマンドの割り当て（allocate-on-demand）とコピー・オン・ライト（copy-on-write）処理が、コンテナ全体の性能に対して影響があるのを理解するのは重要です。

オンデマンドの割り当てが性能に与える影響

devicemapper ストレージ・ドライバは、オンデマンドの割り当て処理時、コンテナに対して新しいブロックを割り当てます。この処理が意味するのは、コンテナの中でアプリケーションが何か書き込みをするごとに、プールか

ら 1 つまたは複数の空ブロックを探し、コンテナの中に割り当てます。

全てのブロックは 64KB です。64KB より小さな書き込みの場合でも、64KB のブロックが 1 つ割り当てられます。これがコンテナの性能に影響を与えます。特にコンテナ内で多数の小さなファイルを書き込む場合に影響があるでしょう。しかしながら、一度ブロックがコンテナに対して割り当てられたら、以降の読み込みは対象のブロックを直接処理できます。

コピー・オン・ライトが性能に与える影響

コンテナ内のデータを初めて更新する度に、毎回 devicemapper ストレージ・ドライバがコピー・オン・ライト処理を行います。このコピーとは、イメージのスナップショット上のデータを、コンテナのスナップショットにコピーするものです。この処理が、コンテナの性能に対して留意すべき影響を与えます。

コピー・オン・ライト処理は 64KB 単位で行います。そのため、1GB のファイルのうち 32KB を更新する場合は、コンテナのスナップショット内にある 64KB のブロックをコピーします。これはファイル・レベルのコピー・オン・ライト処理に比べて、著しい性能向上をもたらします。ファイル・レベルであれば、コンテナ・レイヤに含まれる 1GB のファイル全体をコピーする必要があるからです。

しかしながら、現実的には、コンテナが多くの小さなブロック（64K 以下）に書き込みをするのであれば、devicemapper は AUFS を使うよりも性能が劣ります。

Device Mapper の性能に対するその他の考慮

devicemapper ストレージ・ドライバの性能に対して、他にもいくつかの影響を与える要素があります。

- **動作モード**：Docker が devicemapper ストレージ・ドライバを使用する時、デフォルトのモードは loop-lvm です。このモードはスパース・ファイル（space files；薄いファイル）を使うため、性能を損ないます。そのため、**loop-lvm はプロダクションへのデプロイに推奨されていません** 。プロダクション環境で推奨されるモードは direct-lvm です。これはストレージ・ドライバが直接 raw ブロック・デバイスに書き込みます。
- **高速なストレージ**：ベストな性能を出すためには、 データ・ファイルとメタデータ・ファイルを、 SSD のような高速なストレージ上に配置すべきです。あるいは、SAN や NAS アレイといった、ダイレクト・アタッチ・ストレージでも同様でしょう。
- **メモリ使用量**：Docker ストレージ・ドライバの中で、メモリ使用効率が最も悪いのが devicemapper です。同じコンテナのコピーを n 個起動する時、 n 個のファイルをメモリ上にコピーします。これは、Docker ホスト上のメモリに対して影響があります。そのため、PaaS や他の高密度な用途には、devicemapper ストレージ・ドライバがベストな選択肢とは言えません。

最後に 1 点、データ・ボリュームは最上かつ最も予測可能な性能を提供します。これは、ストレージ・ドライバを迂回し、シン・プロビジョニングやコピー・オン・ライト処理を行わないためです。そのため、データ・ボリューム上で重たい書き込みを行うのに適しています。

5.4.6 OverlayFS ストレージの使用

OverlayFS（オーバレイエフエス） は最近のユニオン・ファイルシステムであり、AUFS に似ています。AUFS と OverlayFS を比較します。

- よりシンプルな設計
- Linux カーネルのバージョン 3.18 からメインラインに取り込まれている
- より速い可能性

この結果、OverlayFS は Docker コミュニティで急に有名になり、多くの人から AUFS の後継者と自然に思われています。OverlayFS は有望ですが、比較的まだ若いストレージ・ドライバです。そのため、プロダクションの

Docker 環境で利用する前に、十分に注意を払うべきでしょう。

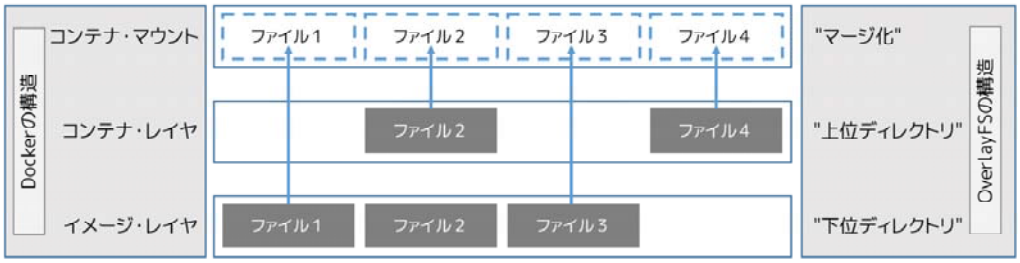
Docker の overlay ストレージ・ドライバは、ディスク上でイメージとコンテナを構築・管理するため、複数の OverlayFS 機能を活用します。

カーネルのメインラインに取り込まれるにあたり、OverlayFS カーネル・モジュール の名前が「overlayfs」から「overlay」に名称変更されました。そのため、同じドキュメントでも 2 つの用語が用いられるかもしれません。ですが、このドキュメントでは「OverlayFS」はファイル全体を指すものとし、overlay は Docker のストレージ・ドライバを指すために遣います。

OverlayFS でイメージのレイヤ化と共有

OverlayFS は 1 つの Linux ホスト上で 2 つのディレクトリを扱います。他のレイヤよりも一番上にあるレイヤにより、1 つに統一して見えます。これらのディレクトリはレイヤとしてたびたび参照され、レイヤに対しては**ユニオン・マウント（union mount）**と呼ばれる技術が使われています。OverlayFS 技術は「下位ディレクトリ」が下の方のレイヤであり、「上位ディレクトリ」が上のレイヤになります。統一して表示される場所そのものを、「マージされた」(marged) ディレクトリと呼びます。

下図は Docker イメージと Docker コンテナがレイヤ化されたものです。イメージ・レイヤは「下位ディレクトリ」であり、コンテナ・レイヤは「上位ディレクトリ」です。統一した表示は「マージされた」ディレクトリであり、これがコンテナに対する事実上のマウント・ポイントです。下図では Docker の構造が OverlayFS 構造にどのように割り当てられているか分かります。



イメージ・レイヤとコンテナ・レイヤに、同じファイルを含められることに注意してください。この時に発生するのは、コンテナ・レイヤ（上位ディレクトリ）に含まれるファイルが優位になり、イメージ・レイヤ（下位ディレクトリ）にある同じファイルを存在しないものとみなします。コンテナ・マウント（マージ化）が、統一した表示をもたらします。

OverlayFS は 2 つのレイヤだけ扱います。つまり、複数にレイヤ化されたイメージは、複数の OverlayFS レイヤとしては使われません。そのかわり、各イメージ・レイヤは /var/lib/docker/overlay ディレクトリ以下で自身が使われます。下位のレイヤと共有するデータを効率的に参照する手法として、ハードリンクが使われます。Docker 1.10 からは、イメージ・レイヤ ID は /var/lib/docker/ 内のディレクトリ名と一致しなくなりました。

コンテナを作成したら、overlay ドライバはコンテナのために新しいディレクトリをイメージの最上位レイヤの上に追加し、これらを組みあわせてディレクトリを表示します。イメージの最上位レイヤは、overlay では「下位ディレクトリ」であり、読み込み専用です。コンテナ用の新しいディレクトリは「上位ディレクトリ」であり、書き込み可能です。

例：イメージとコンテナのディスク上の構造

以下の docker images -a コマンドは、Docker ホスト上の 1 つのイメージを表示しています。表示されているように、イメージは 4 つのレイヤで構成されています。

以下の docker pull コマンドが表しているのは、4 つのレイヤに圧縮された Docker イメージを Docker ホスト上にダウンロードしています。

```
$ sudo docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
8387d9ff0016: Pull complete
3b52deaaf0ed: Pull complete
4bd501fad6de: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:457b05828bdb5dcc044d93d042863fba3f2158ae249a6db5ae3934307c757c54
Status: Downloaded newer image for ubuntu:latest
```

/var/lib/docker/overlay 以下のディレクトリに、各イメージ・レイヤを置くディレクトリがあります。ここが、各イメージ・レイヤの内容を保管する場所です。

以下のコマンドの出力は、取得した各イメージ・レイヤの内容が保管されている 4 つのディレクトリを表しています。しかしながら、これまで見てきたように、イメージ・レイヤ ID は /var/lib/docker/overlay にあるディレクトリ名と一致しません。これは Docker 1.10 以降の通常の挙動です。

```
$ ls -l /var/lib/docker/overlay/
total 24
drwx------ 3 root root 4096 Oct 28 11:02
1d073211c498fd5022699b46a936b4e4bdacb04f637ad64d3475f558783f5c3e
drwx------ 3 root root 4096 Oct 28 11:02
5a4526e952f0aa24f3fcc1b6971f7744eb5465d572a48d47c492cb6bbf9cbcda
drwx------ 5 root root 4096 Oct 28 11:06
99fcaefe76ef1aa4077b90a413af57fd17d19dce4e50d7964a273aae67055235
drwx------ 3 root root 4096 Oct 28 11:01
c63fb41c2213f511f12f294dd729b9903a64d88f098c20d2350905ac1fdbcbba
```

イメージ・レイヤのディレクトリに含まれるファイルはレイヤに対してユニークなものです。つまり、下層レイヤと共有するデータのハード・リンクと同等です。これにより、ディスク容量を効率的に使えます。

また、コンテナは Docker ホストのファイルシステム上の /var/lib/docker/overlay/ 以下に存在します。実行中のコンテナに関するディレクトリを直接 ls -l コマンドで調べたら、次のようなファイルとディレクトリが見えるでしょう。

```
$ ls -l /var/lib/docker/overlay/<実行中コンテナのディレクトリ>
total 16
-rw-r--r-- 1 root root   64 Oct 28 11:06 lower-id
drwxr-xr-x 1 root root 4096 Oct 28 11:06 merged
drwxr-xr-x 4 root root 4096 Oct 28 11:06 upper
drwx------ 3 root root 4096 Oct 28 11:06 work
```

これら 4 つのファイルシステム・オブジェクトは全て OverlayFS が作ったものです。「lower-id」ファイルに含まれるのは、コンテナが元にしたイメージが持つ最上位レイヤの ID です。これは OverlayFS で「lowerdir」（仮想ディレクトリ）として使われます。

```
$ cat
/var/lib/docker/overlay/73de7176c223a6c82fd46c48c5f152f2c8a7e49ecb795a7197c3bb795c4d879e/lower-id
1d073211c498fd5022699b46a936b4e4bdacb04f637ad64d3475f558783f5c3e
```

「upper」（上位）ディレクトリは、コンテナの読み書き可能なレイヤです。コンテナに対するあらゆる変更は、このディレクトリに対して書き込まれます。

「marged」(統合) ディレクトリは効率的なコンテナのマウント・ポイントです。これは、イメージ (「lowerdier」) とコンテナ (「upperdir」) を統合して表示する場所です。あらゆるコンテナに対する書き込みは、直ちにこのディレクトリに反映されます。

「work」(作業) ディレクトリは OverlayFS が機能するために必要です。 コピーアップ (copy_up) 処理などで使われます。

これら全ての構造を確認するには、mount コマンドの出力結果から確認できます (以下の出力は読みやすくするため、省略と改行を施しています)。

```
$ mount | grep overlay
overlay on /var/lib/docker/overlay/73de7176c223.../merged
type overlay (rw,relatime,lowerdir=/var/lib/docker/overlay/1d073211c498.../root,
upperdir=/var/lib/docker/overlay/73de7176c223.../upper,
workdir=/var/lib/docker/overlay/73de7176c223.../work)
```

出力結果から、overlay は読み書き可能 (「rw」) としてマウントされているのが分かります。

overlay でコンテナの読み書き

コンテナのファイル overlay 経由で読み込む、３つのシナリオを考えます。

- **ファイルがコンテナ・レイヤに存在しない場合**。コンテナがファイルを読み込むためにアクセスする時、ファイルがコンテナ (「upperdir」) に存在しなければ、ファイルをイメージ (「lowerdir」) から読み込みます。これにより、非常に小さな性能のオーバーヘッドを生じるかもしれません。
- **ファイルがコンテナ・レイヤのみに存在する場合**。コンテナがファイルを読み込むためにアクセスする時、ファイルがコンテナ (「upperdir」) に存在してイメージ (「lowerdir」) に存在しなければ、コンテナから直接読み込みます。
- **ファイルがコンテナ・レイヤとイメージ・レイヤに存在する場合**。コンテナがファイルを読み込むためにアクセスする時、イメージ・レイヤにもコンテナ・レイヤにもファイルが存在する場合は、コンテナ・レイヤにある方のファイルが読み込まれます。これはコンテナ・レイヤ (「upperdir」) のファイルがイメージ・レイヤ (「lowerdir」) にある同名のファイルを隠蔽するからです。

同様に、コンテナに対するファイルを編集するシナリオを考えましょう。

- **ファイルに対して初めて書き込む場合**。コンテナ上に存在するファイルに初めて書き込む時は、ファイルがコンテナ(「upperdir」)に存在しません。overlay ドライバはコピーアップ処理を行い、イメージ(「lowerdir」)にあるファイルをコンテナ (「upperdir」) にコピーします。コンテナは、以降の書き込みに対する変更は、コンテナ・レイヤ上に新しくコピーしたファイルに対して行います。

しかしながら、OverlayFS はファイル・レベルでの処理であり、ブロック・レベルではありません。つまり、全ての OverlayFS のコピーアップ処理はファイル全体をコピーします。これは、非常に大きなファイルのごく一部分だけを編集する場合でも、全体をコピーします。そのため、コンテナの書き込み性能に対して大きな注意を払う必要があります。

ですが、**次の２つの場合は心配不要です**。

- コピーアップ処理が発生するのは、書き込もうとするファイルを初めて処理する時のみです。以降の書き込み処理は、既にコンテナ上にコピー済みのファイルに対して行われます。
- OverlayFS が動作するのは２つのレイヤのみです。つまり、性能は AUFS より良くなります。AUFS では、多くのイメージ・レイヤがある場合、そこからファイルを探すのに待ち時間発生の考慮が必要

だからです。

- **ファイルとディレクトリを削除する場合**。コンテナ内のファイル削除では、 ホワイトアウト・ファイル（whiteout file）がコンテナ内のディレクトリ（「upperdir」）に作成されます。イメージ・レイヤ（「lowerdir」）にあるバージョンのファイルは削除されません。しかし、コンテナ内のホワイトアウト・ファイルが見えなくします。

コンテナ内のディレクトリを削除したら、「upperdir」で作成されたディレクトリを隠蔽します。これはホワイトアウト・ファイルと同様の効果であり、「lowerdir」イメージのディレクトリを効率的にマスクするものです。

Docker で overlay ストレージ・ドライバを使う設定

Docker が overlay ストレージ・ドライバを使うには、Docker ホスト上の Linux カーネルのバージョンが 3.18（より新しいバージョンが望ましい）であり、overlay カーネル・モジュールを読み込み実行する必要があります。OverlayFS は大部分の Linux ファイルシステムで処理できます。しかし、プロダクション環境での利用にあたっては、現時点では ext4 のみが推奨されています。

以下の手順では Docker ホスト上で OverlayFS を使うための設定方法を紹介します。手順では、Docker デーモンが停止している状態を想定しています。

既に Docker ホスト上で Docker デーモンを使っている場合は、イメージをどこかに保存する必要があります。そのため、処理を進める前に、それらのイメージを Docker Hub やプライベート Docker Trusted Registry に送信しておきます。

1. Docker デーモンが実行中であれば、停止します。

2. カーネルのバージョンと overlay カーネル・モジュールが読み込まれているかを確認します。

```
$ uname -r
3.19.0-21-generic


$ lsmod | grep overlay
overlay
```

3. Docker デーモンを overlay ストレージ・ドライバを使って起動します。

```
$ docker daemon --storage-driver=overlay &
[1] 29403
root@ip-10-0-0-174:/home/ubuntu# INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
INFO[0000] Option DefaultDriver: bridge
INFO[0000] Option DefaultNetwork: bridge
<出力を省略>
```

あるいは、Docker デーモンが自動起動時に必ず overlay ドライバを使うようにします。Docker の設定ファイルを開き、DOCKER_OPTS 行に --storage-driver=overlay フラグを追加します。このオプションを設定しておけば、Docker デーモンを普通に起動するだけで自動的に適用されます。手動で --storage-driver フラグを指定する必要がありません。

4. デーモンが overlay ストレージ・ドライバを使用するのを確認します。

```
$ docker info
Containers: 0
Images: 0
Storage Driver: overlay
 Backing Filesystem: extfs
<出力を省略>
```

この出力では、背後のファイルシステムが extfs なのに注意してください。複数のファイルシステムをサポートしていますが、プロダクションでの使用が推奨されているのは extfs (ext4) のみです。

これで Docker ホストは overlay ストレージ・ドライバを使えるようになりました。mount コマンドを実行したら、Docker が自動的に overlay マウントを作成し、そこに必要となる構成物「lowerdir」「upperdir」「merged」「workdir」も作っています。

OverlayFS と Docker の性能

一般的に overlay ドライバは速いでしょう。aufs と devicemapper と比べれば、ほとんどの場合で速いはずです。特定の環境においては btrfs より速いかもしれません。ここでは、Docker が overlay ストレージ・ドライバを使う時、性能に関して注意すべきことを言及します。

- **ページ・キャッシュ**。OverlayFS はページキャッシュ共有をサポートします。つまり、複数のコンテナが同じファイルにアクセスする時、1 つのページキャッシュ・エントリ（あるいはエントリ群）を共有します。これにより、overlay ドライバはメモリを効率的に使うことができ、PaaS や高密度の使い方に適すでしょう。
- **コピーアップ**。AUFS と同様に、OverlayFS ではコンテナ上のファイルに書き込みする時、初めての場合はコピーアップ処理をします。これは書き込み処理に対して待ち時間を発生させます。特に大きなファイルをコピーアップする場合です。しかし、コピーアップが処理されるのは一度だけであり、以降のファイルに対する書き込みの全てにおいて更なるコピーアップ処理は発生しません。

OverlayFS のコピーアップ処理は AUFS の同じ処理よりも高速でしょう。これは AUFS が OverlayFS より多くのレイヤをサポートしているためであり、多くの AUFS レイヤからファイルを探すのには、時間を必要とする場合があるためです。

- **RPM と Yum**。OverlayFS は POSIX 標準のサブセットのみ実装しています。そのため、いくつかの OverlayFS 処理は POSIX 標準を使っていません。そのような処理の 1 つがコピーアップ処理です。そのため、Docker ホストが overlay ストレージ・ドライバを使っている場合、コンテナの中で yum を使っても動作せず、回避策もありません。
- **i ノード消費**。overlay ストレージ・ドライバの使用は、過度の i ノード費を引き起こします。これは特に Docker ホストが成長し、多くのイメージとコンテナを持つ場合に起こるでしょう。Docker ホストが多くの i ノードを持っていても、コンテナの開始と停止を多く行えば、すぐに i ノードを使い尽くします。

残念ながら、i ノード数を指定できるのはファイルシステムの作成時のみです。そのため、 /var/lib/docker を異なったデバイスにすることを検討した方が良いかもしれません。そのデバイスが自身でファイルシステムを持っており、ファイルシステム作成時に手動で i ノード数を指定する方法があります。

一般的な性能に関するベスト・プラクティスは、OverlayFS にも適用できます。

- **SSD**。ベストな性能のために、SSD（ソリッド・ステート・デバイス）のような高速なストレージ・メディアを使うのは常に良い考えです。

- **データ・ボリュームの使用**。データ・ボリュームは最上かつ最も予測可能な性能を提供します。これは、ストレージ・ドライバを迂回し、シン・プロビジョニングやコピー・オン・ライト処理を行わないためです。そのため、データ・ボリューム上で重たい書き込みを行う場合に使うべきでしょう。

5.4.7 ZFS ストレージの使用

ＺＦＳ（ズィーエフエス） は次世代のファイルシステムです。ボリューム管理、スナップショット、チェックサム処理、圧縮、重複除外（deduplication）、レプリケーションなどの高度なストレージ技術をサポートしています。

ZFS はサン・マイクロシステムズ（現オラクル・コーポレーション）によって開発され、CDDL ライセンスの下でオープンソース化されています。CDDL と GPL 間でライセンスの互換性がないため、ZFS は Linux カーネル・モジュールのメインラインに取り込まれません。しかし、ZFS On Linux（ZoL）プロジェクトにより、外部のカーネル・モジュールとユーザ用のツールを別々にインストールできるように提供されています。

ZFS on Linux（ZoL）への移植は正常かつ成熟しています。しかし ZFS on Linux に対する十分な経験がないのであれば、現時点では `zfs` Docker ストレージ・ドライバをプロダクションで使うことを推奨しません。

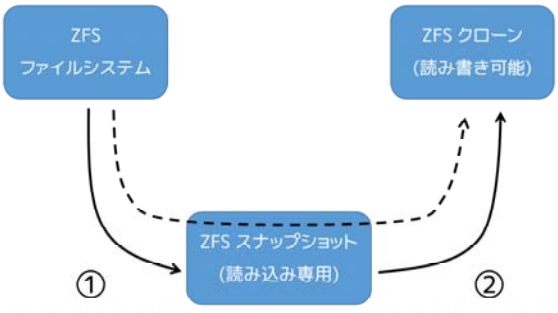
Linux プラットフォーム上では、FUSE（フューズ） で ZFS を実装する方法もあります。Docker とも動作するでしょうが、推奨されません。ネイティブな ZFS ドライバ（ZoL）の方がよりテストされ、性能も良く、より幅広く使われています。このドキュメントでは、ネイティブな ZoL 移植版を言及しているのでご注意ください。

ZFS でイメージのレイヤ化と共有

Docker `zfs` ストレージ・ドライバは、３種類の豊富な ZFS データセットを使えます。

- ファイルシステム（filesystems）
- スナップショット（snapshots）
- クローン（clones）

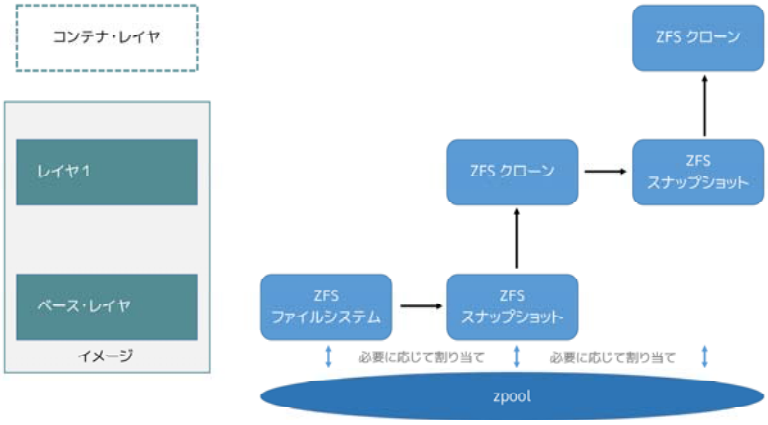
ZFS ファイルシステムは**シン・プロビジョニング**（訳者注：データ書き込みの領域が、初期環境では薄く構築される）であり、**ZFS プール（zpool）**から要求処理があるごとに、領域を割り当てます。スナップショットとクローンは、その時々で ZFS ファイルシステムのコピーをするため、領域を効率的に使います。スナップショットは読み込み専用です。クローンは読み書きできます。クローンはスナップショットからのみ作成可能です。以下の図は、これらの関係性を簡単にしたものです。



図の実線はクローン作成手順の流れです。手順１はファイルシステムのスナップショットを作成します。手順２はスナップショットからクローンを作成します。図の点線はクローンとスナップショットの関係であり、スナップショットを経由しています。

Docker ホストで `zfs` ストレージ・ドライバを使えば、イメージのベース・レイヤは ZFS ファイルシステムにな

ります。それぞれの子レイヤとは、下にあるレイヤの ZFS スナップショットをベースとした ZFS クローンです。コンテナとは、作成するにあたってイメージの最上位レイヤの ZFS スナップショットをベースとした ZFS クローンです。全ての ZFS データセットは、共通の zpool から領域を取り込みます。以下の図は 2 つのレイヤ・イメージをベースにまとめたもので、コンテナを実行しています。



以下で説明する手順は、どのようにイメージをレイヤ化し、コンテナを作成するかです。手順は先ほどの図を元にしています。

1. イメージのベース・レイヤは Docker ホスト上に ZFS ファイルシステムとして存在しています。

ファイルシステムが要領を使うのは、Docker ホスト上のローカル・ストレージ領域 `/var/lib/docker` に zpool を作成する時です。

2. 追加のイメージ・レイヤは、直下にあるイメージ・レイヤが保存されているデータセットのクローンにあたります。

図において、「レイヤ 1」にはベース・レイヤの ZFS スナップショットによって追加されたものです。そして、スナップショットからクローンを作成します。クローンは書き込み可能であり、必要があれば zpool の容量を使います。スナップショットは読み込み専用であり、ベース・レイヤが変更できない（イミュータブルな）ものとして管理されます。

3. コンテナが起動したら、そのイメージ上に読み書き可能なレイヤが追加されます。

先ほどの図では、コンテナの読み書きレイヤは、イメージ（レイヤ 1）の最上位レイヤのスナップショットの上に作成されます。そして、スナップショットからクローンが作成されます。

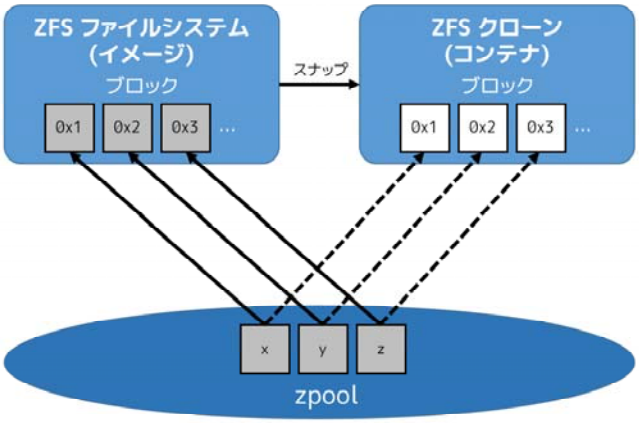
コンテナに対して変更が発生したら、オンデマンドの割り当て（allocate-on-demand）処理によって zpool から領域が割り当てられます。デフォルトでは、ZFS が割り当てる領域は 128KB のブロックです。

子レイヤの作成と、読み込み専用のスナップショットからコンテナを作成する手順において、イメージを不変な（イミュータブルな）オブジェクトとして扱えます。

コンテナを ZFS で読み書き

コンテナが `zfs` ストレージ・ドライバから読み込むのは、非常にシンプルです。直近で起動したコンテナは、ZFS クローンを元にしています。このクローンは作成時、まず全てのデータセットを共有します。つまり、`zfs` ストレージ・ドライバの読み込み処理が高速なことを意味します。これは、読み込み対象のデータがコンテナ内にコピー

されていなくてもです。データブロックの共有は、次のような図になります。



コンテナに対する新規データの書き込みは、オンデマンドの割り当て処理によって完了します。コンテナが書き込みのために新しい領域が必要であれば、その都度、新しいブロックが zpool から割り当てられます。つまりコンテナに対する書き込みによって、新しいデータ用の領域が追加される時のみ、容量を消費します。新しい領域は、根底にある zpool からコンテナ（ZFS クローン）に対して割り当てられるものです。

コンテナ内に存在するデータに対する更新は、コンテナのクローンに新しいブロックを割り当て、変更したデータを新しいブロックに保管したら、処理が完了となります。オリジナルのデータに対する変更は行われません。元になったイメージのデータセットは、変わらない（イミュータブルな）ままです。つまり、通常の ZFS ファイルシステムに対する書き込みと同じであり、そこにコピー・オンライトの仕組みが実装されています。

Docker で ZFS ストレージ・ドライバを使う設定

`zfs` ストレージ・ドライバがサポートされるのは、Docker ホスト上で `/var/lib/docker` が ZFS ファイルシステムでマウントされている場合のみです。このセクションでは、Ubuntu 14.04 システム上に、ネイティブな ZFS on Linux (ZoL) のインストールと設定方法を紹介します。

動作条件

既に Docker ホスト上で Docker デーモンを使っている場合は、イメージをどこかに保存する必要があります。そのため、処理を進める前に、それらのイメージを Docker Hub やプライベート Docker Trusted Registry に送信しておきます。

まず、Docker デーモンを停止します。それから別のブロックデバイス `/dev/xvdb` があることを確認します。このデバイス識別子は皆さんの環境によって異なるかもしれません。そのような場合は、以降の処理で適切なものに置き換えてください。

Ubuntu 14.04 LTS に ZFS をインストール

1. Docker デーモンを実行中であれば、停止します。

2. `software-properties-common` パッケージをインストールします。

この時 `apt-get-repository` コマンドが必要です。

```
$ sudo apt-get install -y software-properties-common
Reading package lists... Done
Building dependency tree
<出力を省略>
```

3. zfs-native パッケージ・アーカイブを追加します。

```
$ sudo add-apt-repository ppa:zfs-native/stable
 The native ZFS filesystem for Linux. Install the ubuntu-zfs package.
<出力を省略>
gpg: key F6B0FC61: public key "Launchpad PPA for Native ZFS for Linux" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
OK
```

4. 全ての登録リポジトリとパッケージ・アーカイブから、最新のパッケージ一覧を取得します。

```
$ sudo apt-get update
Ign http://us-west-2.ec2.archive.ubuntu.com trusty InRelease
Get:1 http://us-west-2.ec2.archive.ubuntu.com trusty-updates InRelease [64.4 kB]
<output truncated>
Fetched 10.3 MB in 4s (2,370 kB/s)
Reading package lists... Done
```

5. ubuntu-zfs パッケージをインストールします。

```
$ sudo apt-get install -y ubuntu-zfs
Reading package lists... Done
Building dependency tree
<出力を省略>
```

6. zfs モジュールを読み込みます。

```
$ sudo modprobe zfs
```

7. 正常に読み込まれていることを確認します。

```
$ lsmod | grep zfs
zfs                  2768247  0
zunicode              331170  1 zfs
zcommon                55411  1 zfs
znvpair                89086  2 zfs,zcommon
spl                    96378  3 zfs,zcommon,znvpair
zavl                   15236  1 zfs
```

ZFS を Docker に設定

ZFS をインストールして読み込んだので、Docker で ZFS 設定をする準備が整いました。

1. 新しい zpool を作成します。

```
$ sudo zpool create -f zpool-docker /dev/xvdb
```

このコマンドは zpool を作成し、そこに「zpool-docker」という名前を割り当てています。この名前は任意です。

2. zpool が存在しているかどうか確認します。

```
$ sudo zfs list
NAME            USED  AVAIL    REFER  MOUNTPOINT
zpool-docker    55K   3.84G    19K    /zpool-docker
```

3. /var/lib/docker に新しい ZFS ファイルシステムを作成・マウントします。

```
$ sudo zfs create -o mountpoint=/var/lib/docker zpool-docker/docker
```

4. 直前の手順が正常に行われたか確認します。

```
$ sudo zfs list -t all
NAME                USED  AVAIL  REFER  MOUNTPOINT
zpool-docker        93.5K  3.84G    19K  /zpool-docker
zpool-docker/docker  19K    3.84G    19K  /var/lib/docker
```

これで ZFS ファイルシステムを /var/lib/docker にマウントしました。デーモンは自動的に zfs ストレージを読み込むでしょう。

5. Docker デーモンを起動します。

```
$ sudo service docker start
docker start/running, process 2315
```

使用している Linux ディストリビューションによっては、この Docker デーモンの開始手順は少し異なる場合があります。Docker デーモンに対して zfs ストレージ・ドライバの利用を明示する場合は、 docker daemon コマンドで --storage-driver=zfs フラグを使うか、Docker 設定ファイル中の DOCKER_OPTS 行を編集します。

6. デーモンが zfs ストレージ・ドライバを使っているのを確認します。

```
$ sudo docker info
Containers: 0
Images: 0
Storage Driver: zfs
 Zpool: zpool-docker
 Zpool Health: ONLINE
 Parent Dataset: zpool-docker/docker
 Space Used By Parent: 27648
 Space Available: 4128139776
 Parent Quota: no
 Compression: off
Execution Driver: native-0.2
[...]
```

先ほどの出力は、Docker デーモンが zfs ストレージ・ドライバを使っており、親データセットは先ほど作成した zpool-docker/docker ファイルシステムだと分かります。

これで Docker ホストは、イメージとコンテナの管理・保管に ZFS を使います。

ZFS と Docker 性能

Docker で zfs ストレージ・ドライバを使うにあたり、パフォーマンスに影響を与えるいくつかの要素があります。

- **メモリ**。ZFS の性能に、メモリはとても大きな影響があります。そもそもの事実として、本来の ZFS は、大きな Sun Solaris サーバ上で大容量のメモリを使うよう設計されていました。Docker ホストのサイジング時には、この点を忘れないでください。
- **ZFS の機能**。ZFS の機能、例えば重複削除（deduplication）は ZFS が使うメモリ容量が明らかに増加します。メモリの消費と性能面から、ZFS 重複削除の機能を無効にすることを推奨します。しかし、別のスタック上（SAN や NAS アレイ）のレイヤに対する重複削除は、ZFS のメモリ使用や性能に関する影響がありませんので、利用できるでしょう。もし SAN、NAS、その他のハードウェア RAID 技術を使うのであれば、ZFS の利用にあたり、以下にある既知のベストプラクティスをご利用ください。
- **ZFS キャッシュ**。ZFS はディスク・ブロックを、アダプティブ・リプレースメント・キャッシュ（ARC; adaptive replacement cache）と呼ばれるメモリ上の構造にキャッシュします。ZFS の Single Copy ARC 機能により、ブロックをコピーした単一キャッシュが、ファイルシステムの複数のクローンから共有されます。つまり、複数の実行中のコンテナは、キャッシュされたブロックのコピーを共有できるのです。これが意味するのは、ZFS は PaaS や他の高密度の利用例にとっては良い選択肢となるでしょう。
- **断片化**。断片化は ZFS のようなコピー・オン・ライトなファイルシステムにおける、自然な副産物です。ZFS は 128KB のブロックに書き込みますが、slabs（複数の 128KB ブロック）をコピー・オン・ライト処理に割り当てますので、断片化を減らそうとしています。また、ZFS intent log（ZIL）と書き込みの一体化も断片化を減らすものです。
- **ネイティブな Linux 用 ZFS ドライバの使用**。Docker zfs ストレージ・ドライバは ZFS FUSE 実装をサポートしているとはいえ、高い性能が必要な場合は推奨されません。ネイティブな Linux 用 ZFS ドライバは、FUSE 実装よりも良い性能でしょう。

以下の一般的な性能に関するベストプラクティスは、ZFS でも適用できます。

- **SSD**。ベストな性能のために、SSD（ソリッド・ステート・デバイス）のような高速なストレージ・メディアを使うのは常に良い考えです。十分に利用可能な SSD ストレージ容量があるのなら、ZIL を SSD 上に置くことを推奨します。
- **データ・ボリュームの使用**。データ・ボリュームは最上かつ最も予測可能な性能を提供します。これは、ストレージ・ドライバを迂回し、シン・プロビジョニングやコピー・オン・ライト処理を行わないためです。そのため、データ・ボリューム上で重たい書き込みを場合に使うべきでしょう。

5.5 ネットワーク設定

5.5.1 Docker ネットワーク機能の概要

このセクションでは、どのようにして Docker のネットワーク機能を使うかを説明します。ネットワーク機能により、ユーザは自分自身でネットワークを定義し、コンテナを接続できます。他にも、単一のホスト上でのネットワーク作成や、複数のホストを横断するネットワークを作成できます。

デフォルトのブリッジ・ネットワーク bridge0（ブリッジゼロ）に慣れている方向けに、旧来のネットワーク機能のサポートが続きます。Docker はインストール時にネットワークを自動的作成します。おの、デフォルト・ブリッジ・ネットワークを bridge（ブリッジ）と呼びます。このネットワークに関連する情報は、Docke デフォルト・ブリッジ・ネットワークのセクションをご覧ください。

5.5.2 Docker コンテナ・ネットワークの理解

ウェブ・アプリケーションの構築は、安全についての考慮が必要であり、そのために Docker ネットワーク機能を使います。ネットワークとは、定義上、コンテナのために完全な**分離**（**isolation**（アイソレーション））を提供するものです。そして、アプリケーションの実行にあたり、ネットワーク管理は重要であることを意味します。Docker コンテナ・ネットワークは、これらを管理するものです。

このセクションでは、Docker Engine ドライバ固有の標準ネットワーク機能について、その概要を扱います。ここでは標準のネットワーク・タイプについてと、どのようにして自分自身でユーザ定義ネットワークを使うのかを説明します。また、単一ホストまたはクラスタ上をまたがるホスト間で、ネットワークを作成するために必要なリソースについても説明します。

デフォルト・ネットワーク

Docker のインストールは、自動的に３つのネットワークを作成します。ネットワーク一覧を表示するには **docker network ls**（ネットワークエルエス）コマンドを使います。

```
$ docker network ls
NETWORK ID          NAME                DRIVER
7fca4eb8c647        bridge              bridge
9f904ee27bf5        none                null
cf03ee007fb4        host                host
```

これまで、３つのネットワークが Docker の一部として実装されました。コンテナを実行するネットワークは --net フラグで指定できます。それでも、これらの３つのネットワークは今も利用可能です。

Docker をインストールした全ての環境には、docker0（ドッカーゼロ）と表示される**ブリッジ** (**bridge**) ネットワークが現れます。オプションで docker run --net=<ネットワーク名> を指定しない限り、Docker デーモンはデフォルトでこのネットワークにコンテナを接続します。ホスト上で ifconfig コマンドを使えば、ホストネットワーク上のスタックの一部として、このブリッジを見ることができます。

```
ubuntu@ip-172-31-36-118:~$ ifconfig
docker0   Link encap:Ethernet  HWaddr 02:42:47:bc:3a:eb
          inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:47ff:febc:3aeb/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:9001  Metric:1
          RX packets:17 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

```
          RX bytes:1100 (1.1 KB)  TX bytes:648 (648.0 B)
```

コンテナにネットワーク層を追加しない場合は、 none ネットワークを指定します。そのコンテナはネットワーク・インターフェースが欠如します。コンテナに**接続（アタッチ）**すると、次のようなネットワーク情報を表示します。

```
ubuntu@ip-172-31-36-118:~$ docker attach nonenetcontainer

/ # cat /etc/hosts
127.0.0.1    localhost
::1  localhost ip6-localhost ip6-loopback
fe00::0      ip6-localnet
ff00::0      ip6-mcastprefix
ff02::1      ip6-allnodes
ff02::2      ip6-allrouters
/ # ifconfig
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ #
```

コンテナから離れても実行したままにするには、CTRL-p CTRL-q を押します。

今度はホスト側の（host）ネットワーク・スタックにコンテナに接続します。コンテナ内のネットワーク設定が、ホスト上と同じに見えるでしょう。

bridge ネットワークの使用時をのぞけば、これらのデフォルト・ネットワークと実際に通信する必要はありません。このように一覧を表示したり調べたりできますが、削除できません。これらは Docker の導入に必要だからです。一方、ユーザ定義ネットワークであれば自分で追加、あるいは必要がなければ削除できます。自分でネットワークを作る前に、デフォルトのネットワークについて少しだけ見ておきましょう。

デフォルトのブリッジ・ネットワーク詳細

Docker ホスト上の全てのデフォルト・ネットワーク・ブリッジを表示するには、docker network inspect を使います。

```
$ docker network inspect bridge
[
   {
       "Name": "bridge",
       "Id": "f7ab26d71dbd6f557852c7156ae0574bbf62c42f539b50c8ebde0f728a253b6f",
       "Scope": "local",
       "Driver": "bridge",
       "IPAM": {
           "Driver": "default",
           "Config": [
               {
```

```
                "Subnet": "172.17.0.1/16",
                "Gateway": "172.17.0.1"
            }
        ]
    },
    "Containers": {},
    "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": "9001"
    }
  }
]
```

Docker Engine は自動的にネットワークの**サブネット**(**Subnet**)と**ゲートウェイ**(**Gateway**)を作成します。docker run コマンドは新しいコンテナに対して、自動的にこのネットワークを割り当てます。

```
$ docker run -itd --name=container1 busybox
3386a527aa08b37ea9232cbcace2d2458d49f44bb05a6b775fba7ddd40d8f92c

$ docker run -itd --name=container2 busybox
94447ca479852d29aeddca75c28f7104df3c3196d7b6d83061879e339946805c
```

２つのコンテナを実行してから、再びこのブリッジ・ネットワークを参照し、直近のコンテナのネットワークがどのようになっているか見てみましょう。コンテナの ID が表示されるようになります。

```
$ docker network inspect bridge
{[
    {
        "Name": "bridge",
        "Id": "f7ab26d71dbd6f557852c7156ae0574bbf62c42f539b50c8ebde0f728a253b6f",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.17.0.1/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Containers": {
            "3386a527aa08b37ea9232cbcace2d2458d49f44bb05a6b775fba7ddd40d8f92c": {
                "EndpointID": "647c12443e91faf0fd508b6edfe59c30b642abb60dfab890b4bdccee38750bc1",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            },
            "94447ca479852d29aeddca75c28f7104df3c3196d7b6d83061879e339946805c": {
                "EndpointID": "b047d090f446ac49747d3c37d63e4307be745876db7f0ceef7b311cbba615f48",
```

```
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "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": "9001"
        }
    }
]
```

上の docker network inspect コマンドは、接続しているコンテナと特定のネットワーク上にある各々のネットワークを全て表示します。デフォルト・ネットワークのコンテナは、IP アドレスを使って相互に通信できます。デフォルトのネットワーク・ブリッジ上では、Docker は自動的なサービス・ディスカバリをサポートしていません。このデフォルト・ブリッジ・ネットワーク上でコンテナ名を使って通信をしたい場合、コンテナ間の接続には過去の docker run --link オプションを使う必要があります。

実行しているコンテナに接続（attach）すると、設定を調査できます。

```
$ docker attach container1

/ # ifconfig
ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:9001  Metric:1
          RX packets:16 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1296 (1.2 KiB)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

この bridge ネットワークにおけるコンテナの接続性をテストするため、３秒間 ping を実行します。

```
/ # ping -w3 172.17.0.3
PING 172.17.0.3 (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.096 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.17.0.3: seq=2 ttl=64 time=0.074 ms

--- 172.17.0.3 ping statistics ---
```

```
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.074/0.083/0.096 ms
```

最後に cat コマンドを使い、container1 のネットワーク設定を確認します。

```
/ # cat /etc/hosts
172.17.0.2   3386a527aa08
127.0.0.1    localhost
::1  localhost ip6-localhost ip6-loopback
fe00::0      ip6-localnet
ff00::0      ip6-mcastprefix
ff02::1      ip6-allnodes
ff02::2      ip6-allrouters
```

container1 からデタッチするには、CTRL-p　CTRL-q を使って離れます。それから container2 にアタッチし、３つのコマンドを繰り返します。

```
$ docker attach container2

/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:03
          inet addr:172.17.0.3  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:9001  Metric:1
          RX packets:15 errors:0 dropped:0 overruns:0 frame:0
          TX packets:13 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1166 (1.1 KiB)  TX bytes:1026 (1.0 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ # ping -w3 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.067 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.075 ms
64 bytes from 172.17.0.2: seq=2 ttl=64 time=0.072 ms

--- 172.17.0.2 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.067/0.071/0.075 ms
/ # cat /etc/hosts
172.17.0.3   94447ca47985
127.0.0.1    localhost
::1  localhost ip6-localhost ip6-loopback
fe00::0      ip6-localnet
ff00::0      ip6-mcastprefix
ff02::1      ip6-allnodes
ff02::2      ip6-allrouters
```

デフォルトの docker0 ブリッジ・ネットワークは、ポート・マッピング（割り当て）機能の使用と、docker run --link によって docker0 ネットワーク上にあるコンテナ間の通信を可能とします。これらの技術はセットアップが面倒であり、間違いしがちです。この技術はまだ利用可能ですが、これらを使わず、その代わりに自分自身でブリッジ・ネットワークを定義するのが望ましいです。

ユーザ定義ネットワーク

コンテナのより優れた分離のために、自分で**ユーザ定義ネットワーク**(user-defined network)を作成できます。Docker はこれらネットワークを作成するための、複数の**ネットワーク・ドライバ**を標準提供しています。新しいブリッジ・ネットワークやオーバレイ・ネットワークを作成できます。また、自分でネットワーク・プラグインを書き、リモート・ネットワークを定義できます。

ネットワークは複数作成できます。コンテナを１つ以上のネットワークに追加できます。コンテナの通信はネットワーク内だけでなく、ネットワーク間を横断できます。コンテナが２つのネットワークにアタッチする時、どちらのネットワークに対しても通信可能です。

以降のセクションでは、各 Docker 内蔵ネットワーク・ドライバに関するより詳細を扱います。

ブリッジ・ネットワーク

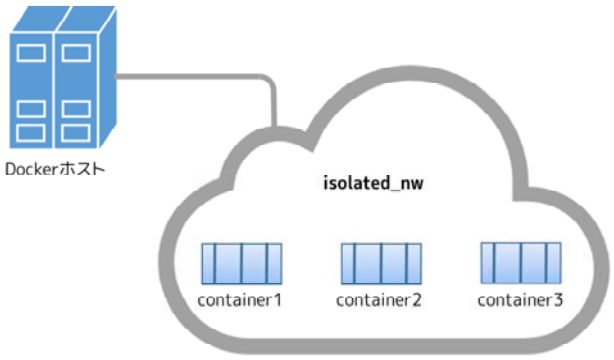
最も簡単なユーザ定義ネットワークは、ブリッジ・ネットワークの作成です。このネットワークは過去の docker0 ネットワークと似ています。いくつかの新機能が追加されていますが、古い機能のいくつかは利用できません。

```
$ docker network create --driver bridge isolated_nw
1196a4c5af43a21ae38ef34515b6af19236a3fc48122cf585e3f3054d509679b

$ docker network inspect isolated_nw
[
    {
        "Name": "isolated_nw",
        "Id": "1196a4c5af43a21ae38ef34515b6af19236a3fc48122cf585e3f3054d509679b",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.21.0.0/16",
                    "Gateway": "172.21.0.1/16"
                }
            ]
        },
        "Containers": {},
        "Options": {}
    }
]

$ docker network ls
NETWORK ID          NAME                DRIVER
9f904ee27bf5        none                null
cf03ee007fb4        host                host
7fca4eb8c647        bridge              bridge
c5ee82f76de3        isolated_nw         bridge
```

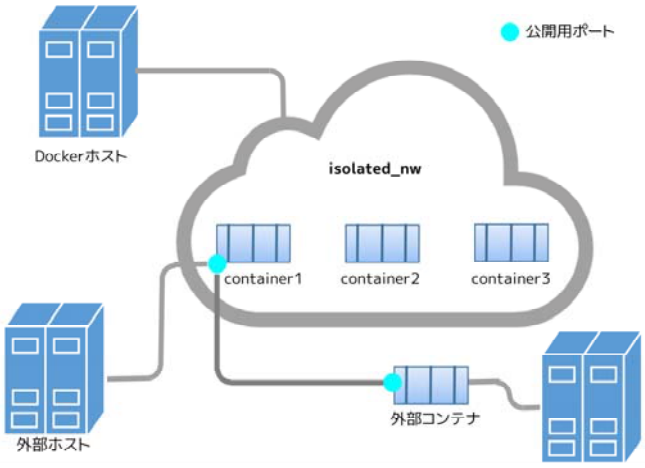
ネットワークを作成したら、コンテナ起動時に docker run --net=<ネットワーク名> オプションを指定して接続できます。

```
$ docker run --net=isolated_nw -itd --name=container3 busybox
885b7b4f792bae534416c95caa35ba272f201fa181e18e59beba0c80d7d77c1d

$ docker network inspect isolated_nw
[
    {
        "Name": "isolated_nw",
        "Id": "1196a4c5af43a21ae38ef34515b6af19236a3fc48122cf585e3f3054d509679b",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {}
            ]
        },
        "Containers": {
            "885b7b4f792bae534416c95caa35ba272f201fa181e18e59beba0c80d7d77c1d": {
                "EndpointID": "514e1b419074397ea92bcfaa6698d17feb62db49d1320a27393b853ec65319c3",
                "MacAddress": "02:42:ac:15:00:02",
                "IPv4Address": "172.21.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {}
    }
]
```

このネットワーク内で起動したコンテナは、Docker ホスト上の他のコンテナとは独立しています。ネットワーク内の各コンテナは速やかに通信が可能です。しかし、コンテナ自身が含まれるネットワークは外部のネットワークから独立しています。



ユーザ定義ブリッジ・ネットワークの内部では、リンク機能はサポートされません。ですが、このネットワーク上にあるコンテナのポートは公開可能です。bridge ネットワークの一部を外のネットワークから使う時に便利でしょう。

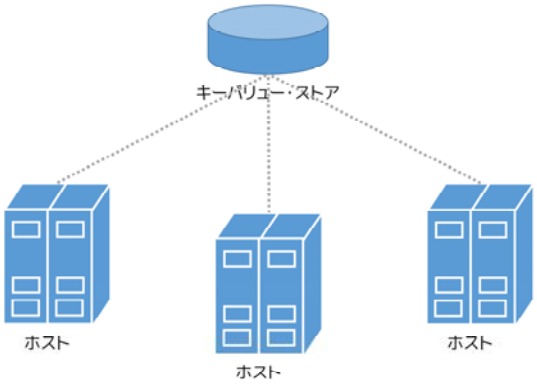


ブリッジ・ネットワークは、単一ホスト上で比較的小さなネットワークの実行時に便利です。それだけではありません。**オーバレイ・ネットワーク**を使うと更に大きなネットワークを作成できます。

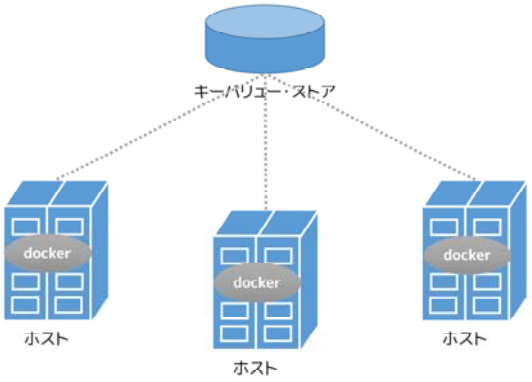
オーバレイ・ネットワーク

Docker の**オーバレイ（overlay）ネットワーク・ドライバ**は、複数ホストのネットワーキングにネイティブに対応する革新的なものです。この機能のサポートは `libnetwork`、VXLAN を基盤とした内部オーバレイ・ネットワーク・ドライバ、そして Docker の `libkv` ライブラリによる成果です。

オーバレイ・ネットワークはキーバリュー・ストア・サービスが必要です。現時点で Docker の `libkv` がサポートしているのは、Consul、Etcd、Zookeeper（分散ストア）です。ネットワークを作成する前に、キーバリュー・ストア・サービスを選び、設定する必要があります。そして、Docker ホスト側では、ネットワークとサービスを通信できるようにします。



ネットワークの各ホストは、それぞれで Docker エンジンを動かす必要があります。最も簡単なのは Docker Machine を使ってホストをプロビジョン[*1] する方法です。



*1 訳者注：provision = ホスト環境の自動構築と、OS の自動セットアップ作業を指します。

ホスト間で以下のポートをオープンにすべきです。

プロトコル	Port	説明
udp	4789	データ用（VXLAN）
tcp/udp	7946	管理用

使用するキーバリュー・ストアによっては、追加ポートが必要になる場合があります。各ベンダーのドキュメントを確認し、必要なポートを開いてください。

Docker Machine でプロビジョニングしたら、Docker Swarm（スウォーム）を使うための Swarm とディスカバリ・サービスも同様に迅速に入れられます。

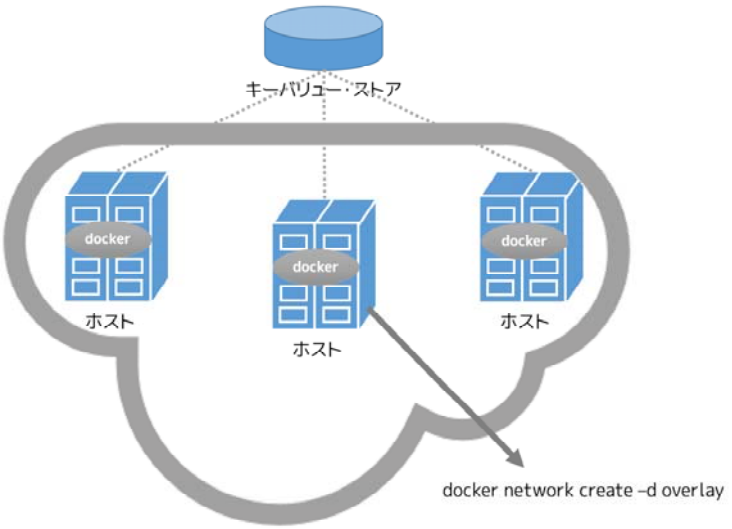
オーバレイ・ネットワークを作成するには、各々の Docker Engine 上のデーモンのオプションで、overlay ネットワークを設定します。そこには２つの設定オプションがあります：

オプション	説明
--cluster-store=プロバイダ://URL	キーバリュー・サービスの場所を指定
--cluster-advertise=HOST_IP|HOST_IFACE:PORT	クラスタ用に使うホストのインターフェース用 IP アドレス
--cluster-store-opt=KVS	KVS のオプション。TLS 証明書やディスカバリ間隔の調整のようなオプション。

overlay ネットワークを Swarm のマシン上に作成します。

```
$ docker network create --driver overlay my-multi-host-network
```

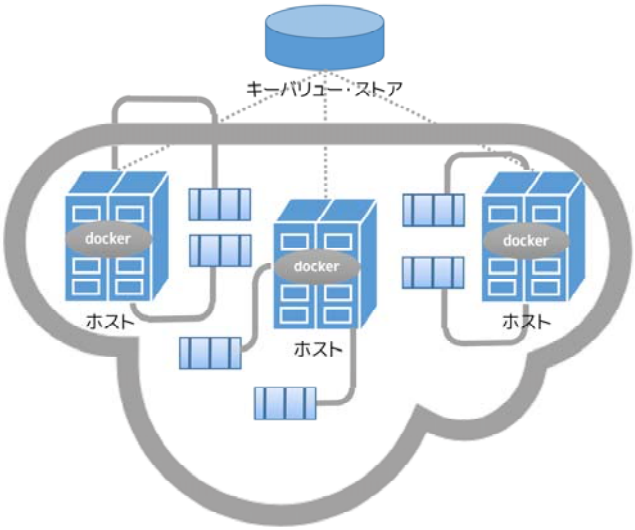
この結果、複数のホストを横断する１つのネットワークができます。 overlay ネットワークはコンテナに対して、完全なる独立機能を提供します。



以後、各ホスト上でコンテナ起動時にこのネットワーク名を指定します。

```
$ docker run -itd --net=my-multi-host-network busybox
```

接続したあと、ネットワーク内の各コンテナ全てにアクセス可能となります。この時、コンテナがどこのホスト上で起動しているか気にする必要はありません。



自分で試したい場合は、Docker Machine の overlay 導入ガイドをご覧ください。

カスタム・ネットワーク・プラグイン

必要があれば、自分自身でネットワーク・ドライバ・プラグインを書けます。ネットワーク・ドライバ・プラグインは Docker のプラグイン基盤を使います。この基盤を使い、Docker デーモン が動作する同じ Docker ホストでプラグインをプロセスとして実行します。

ネットワーク・プラグインは、他のプラグインと同様、いくつかの制約やインストール時のルールがあります。全てのプラグインはプラグイン API を利用します。これらはインストール、開始、停止、有効化といったライフサイクル全般に及びます。

カスタム・ネットワーク・ドライバを作成してインストールした後は、内部ネットワーク・ドライバと同じように扱えます。例：

```
$ docker network create --driver weave mynet
```

必要があれば自分で内部を確認できますし、更なるコンテナを追加など、いろいろ可能ます。もちろん、プラグインごとに異なった技術やフレームワークを使うこともあるでしょう。カスタム・ネットワークは Docker の標準ネットワークが持たない機能を実装できます。プラグインの書き方に関する詳細情報は、Docker 拡張のセクション、またはネットワーク・ドライバ・プラグインの書き方をお読みください。

Docker 内部 DNS サーバ

Docker デーモンは**内部 DNS サーバ（embedded DNS server）**を動かし、ユーザ定義ネットワーク上でコンテナがサービス・ディスカバリを自動的に行えるようにします。コンテナから名前解決のリクエストがあれば、内部 DN サーバを第一に使います。リクエストがあっても内部 DNS サーバが名前解決できなければ、外部の DNS サーバにコンテナからのリクエストを転送します。割り当てできるのはコンテナの作成時だけです。内部 DNS サーバが到達可能なのは 127.0.0.11 のみであり、コンテナの resolv.conf に書かれます。ユーザ定義ネットワーク上の内部 DNS サーバに関しては ユーザ定義ネットワーク用の内部 DNS サーバ をご覧ください。

リンク機能

Docker ネットワーク機能より以前は、Docker リンク機能を使いコンテナの相互発見や、特定のコンテナから別のコンテナに安全に情報を送信できました。Docker ネットワークを導入したら、自動的にコンテナを名前で発見できます。しかし、デフォルトの `docker0` ブリッジ・ネットワークとユーザ定義ネットワークには違いがあるため、

まだリンク機能を使うこともできます。より詳しい情報については、 「古いリンク機能のデフォルト bridge ネットワークのリンク機能」のセクションをご覧ください。ユーザ定義ネットワークでリンク機能を使うには「ユーザ定義ネットワークでコンテナをリンク」のセクションをご覧ください。

5.5.3 network コマンドを使う

このセクションではネットワーク・サブコマンドの例を扱います。このサブコマンドは Docket ネットワークを相互に扱い、コンテナをネットワークに配置します。コマンドは Docker エンジン CLI を通して利用可能です。コマンドとは以下の通りです。

- docker network create
- docker network connect
- docker network ls
- docker network rm
- docker network disconnect
- docker network inspect

このセクションの例に取り組む前に、 Docker ネットワークの理解を読むのは良い考えです。なお、この例では bridge ネットワークを使用するため、すぐに試せます。overlay ネットワークを試したいのであれば、「マルチホスト・ネットワーキングを始める」のセクションをご覧ください。

ネットワークの作成

Docker Engine をインストールしたら、Docker Engine は自動的に **bridge ネットワーク**を作成します。このネットワークとは、Docker Engine が従来使ってきた **docker0 ブリッジ**に相当します。このデフォルトのネットワークだけでなく、自分自身で**ブリッジ（bridge）ネットワーク**や**オーバレイ（overlay）ネットワーク**を作成可能です。

bridge ネットワークは Docker エンジンの実行ホスト環境上に存在します。overlay ネットワークは、複数のホスト上で動くエンジンを横断します。docker network（ネットワーク） create（クリエイト） を実行する時、ネットワーク名だけ指定したら、自動的にブリッジ・ネットワークを作成します。

```
$ 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": {}
    }
```

]

overlay ネットワークの場合は、ブリッジ・ネットワークとは異なります。作成前にいくつかの事前準備が必要です。事前準備は次の項目です。

- キーバリュー・ストアへのアクセス。エンジンがサポートするキーバリュー・ストアは Consul、Etcd、ZooKeeper（分散ストア）です。
- ホストのクラスタが、キーバリュー・ストアへ接続できること。
- 各ホスト上のエンジン daemon に、 Swarm クラスタとしての適切な設定をすること。

overlay ネットワークがサポートする docker daemon のオプションは、次の通りです。

- --cluster-store
- --cluster-store-opt
- --cluster-advertise

また、必要がなくてもクラスタ管理用に Docker　Swarm をインストールするのも良い考えでしょう。Swarm はクラスタの設定を手助けするために、洗練されたディスカバリとサーバ管理機能を持っています。

ネットワーク作成時、Docker Engine はデフォルトでサブネットが重複しないネットワークを作成します。このデフォルトの挙動は変更できます。特定のサブネットワークを直接指定するには --subnet（サブネット） オプションを使います。 bridge ネットワーク上では 1 つだけサブネットを作成できます。 overlay ネットワークでは、複数のサブネットをサポートしています。

ネットワークの作成時は --subnet オプションの指定を強く推奨します。 --subnet を指定しなければ、docker デーモンはネットワークに対してサブネットを自動的に割り当てます。その時、Docker が管理していない基盤上の別サブネットと重複する可能性が有り得ます。このような重複により、コンテナがネットワークに接続する時に問題や障害を引き起こします。

--subnet オプションの他にも、 --gateway（ゲートウェイ）、--ip-range（アイピーレンジ）、 --aux-address（オーアドレス） オプションが指定可能です。

```
$ 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注意してください。重複したらネットワーク作成が失敗し、Docker Engine はエラーを返します。

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）は追加オプションを指定できます。dokcer0 ブリッジにおいては、Docker デーモンのフラグで指定するのと同等の以下の設定が利用できます。

オプション	同等	説明
com.docker.network.bridge.name	—	Linux ブリッジ作成時に使うブリッジ名
com.docker.network.bridge.enable_ip_masquerade	--ip-masq	IP マスカレードを有効化
com.docker.network.bridge.enable_icc	--icc	Docker 内部におけるコンテナの接続性を有効化・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（割り当てる）デフォルトの IP
com.docker.network.mtu	--mtu	コンテナのネットワーク MTU を設定

docker network create 実行時、次の引数をあらゆるネットワーク・ドライバで指定できます。

引数	同等	説明
--internal	—	ネットワークから外部へのアクセスを制限
--ipv6	--ipv6	IPv6 ネットワーク機能の有効化

例えば、 -o または --opt オプションを使い、ポートを公開用に割り当てる IP アドレスを指定しましょう。

```
$ docker network create -o "com.docker.network.bridge.host_binding_ipv4"="172.23.0.1" my-network
b1a086897963e6a2e7fc6868962e55e746bee8ad0c97b54a5831054b5f62672a
$ docker network inspect my-network
[
    {
        "Name": "my-network",
        "Id": "b1a086897963e6a2e7fc6868962e55e746bee8ad0c97b54a5831054b5f62672a",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.23.0.0/16",
                    "Gateway": "172.23.0.1/16"
                }
            ]
        },
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.host_binding_ipv4": "172.23.0.1"
        }
    }
]
$ docker run -d -P --name redis --net my-network redis
bafb0c808c53104b2c90346f284bda33a69beadcab4fc83ab8f2c5a4410cd129
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
  PORTS                           NAMES
bafb0c808c53        redis               "/entrypoint.sh redis"   4 seconds ago       Up 3 seconds
 172.23.0.1:32770->6379/tcp   redis
```

コンテナに接続

コンテナは 1 つまたは複数のネットワークに対して、動的に接続できます。これらのネットワークは、同じネットワーク・ドライバの場合もあれば、異なるバックエンドの場合もあります。接続後は、コンテナから他のコンテナに IP アドレスまたはコンテナ名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・プラグインの場合は、複数のホストへの接続性をサポートしており、コンテナは同一ホストで作成されたマルチホスト・ネットワークだけでなく、異なったホスト上で作成された環境とも同様に通信可能です。

ここでは例として、２つのコンテナを作成します。

```
$ docker run -itd --name=container1 busybox
18c062ef45ac0c026ee48a83afa39d25635ee5f02b58de4abc8f467bcaa28731

$ docker run -itd --name=container2 busybox
498eaaaf328e1018042c04b2de04036fc04719a6e39a097a4f4866043a2c2152
```

それから、分離用の bridge ネットワークを作成します。

```
$ docker network create -d bridge --subnet 172.25.0.0/16 isolated_nw
06a62f1c73c4e3107c0f555b7a5f163309827bfbbf999840166065a8f35455a8
```

このネットワークに container2 を追加し、ネットワークへの接続性を調査（ inspect ）します。

```
$ docker network connect isolated_nw container2
$ docker network inspect isolated_nw
[[
    {
        "Name": "isolated_nw",
        "Id": "06a62f1c73c4e3107c0f555b7a5f163309827bfbbf999840166065a8f35455a8",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.21.0.0/16",
                    "Gateway": "172.21.0.1/16"
                }
            ]
        },
        "Containers": {
            "90e1f3ec71caf82ae776a827e0712a68a110a3f175954e5bd4222fd142ac9428": {
                "Name": "container2",
                "EndpointID": "11cedac1810e864d6b1589d92da12af66203879ab89f4ccd8c8fdaa9b1c48b1d",
                "MacAddress": "02:42:ac:19:00:02",
                "IPv4Address": "172.25.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {}
    }
]
```

Docker Engine が container2 対して、自動的に IP アドレスを割り当てているのが分かります。もしもネットワ

ーク作成時に --subnet を指定していたならば、Docker Engine は指定されたサブネットから IP アドレスを取得します。次に 3 つめのコンテナを起動します。このネットワークにコンテナを接続するには、 docker run コマンドで --net オプションを使います。

```
$ docker run --net=isolated_nw --ip=172.25.3.3 -itd --name=container3 busybox
467a7863c3f0277ef8e661b38427737f28099b61fa55622d6c30fb288d88c551
```

見ての通り、コンテナに対して IP アドレスを指定できました。docker run コマンドでコンテナ作成時に、ユーザが接続先のサブネットを指定したら、任意の IPv4 アドレスと同時、あるいは別に IPv6 アドレスも指定できます。また、 docker network connect コマンドでも追加できます。IP アドレスの指定は、コンテナのネットワーク設定の一部です。そのため、コンテナを再起動しても IP アドレスは維持されるでしょう。将来的にはユーザ定義ネットワーク上でのみ利用可能になります。ユーザ定義ネットワーク以外では、デーモン再起動時にサブネット設定情報の維持を保証しないためです。

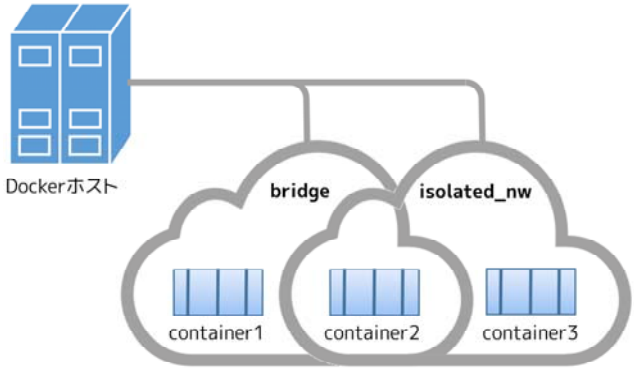
次は、container3 に対するネットワークのリソースを調査します。

```
$ docker inspect --format='{{json .NetworkSettings.Networks}}'  container3
{"isolated_nw":{"IPAMConfig":{"IPv4Address":"172.25.3.3"},"NetworkID":"1196a4c5af43a21ae38ef34515b6a
f19236a3fc48122cf585e3f3054d509679b",
"EndpointID":"dffc7ec2915af58cc827d995e6ebdc897342be0420123277103c40ae35579103","Gateway":"172.25.0.
1","IPAddress":"172.25.3.3","IPPrefixLen":16,"IPv6Gateway":"","GlobalIPv6Address":"","GlobalIPv6Pref
ixLen":0,"MacAddress":"02:42:ac:19:03:03"}}
```

このコマンドを container2 にも繰り返します。Python をインストール済みであれば、次のように表示を分かりやすくできるでしょう。

```
$ docker inspect --format='{{json .NetworkSettings.Networks}}'  container2 | python -m json.tool
{
    "bridge": {
        "NetworkID":"7ea29fc1412292a2d7bba362f9253545fecdfa8ce9a6e37dd10ba8bee7129812",
        "EndpointID": "0099f9efb5a3727f6a554f176b1e96fca34cae773da68b3b6a26d046c12cb365",
        "Gateway": "172.17.0.1",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "IPAMConfig": null,
        "IPAddress": "172.17.0.3",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "MacAddress": "02:42:ac:11:00:03"
    },
    "isolated_nw": {
        "NetworkID":"1196a4c5af43a21ae38ef34515b6af19236a3fc48122cf585e3f3054d509679b",
        "EndpointID": "11cedac1810e864d6b1589d92da12af66203879ab89f4ccd8c8fdaa9b1c48b1d",
        "Gateway": "172.25.0.1",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "IPAMConfig": null,
        "IPAddress": "172.25.0.2",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "MacAddress": "02:42:ac:19:00:02"
    }
}
```

container2 は２つのネットワークに所属しているのが分かります。bridge ネットワークは起動時にデフォルトで参加したネットワークであり、isolated_nw ネットワークは後から自分で接続したものです。



container3 の場合、docker run では isolated_nw に接続しました。そのため、このコンテナは bridge に接続していません。

docker attach コマンドで実行中の container2 に接続し、ネットワーク・スタックを確認しましょう。

```
$ docker attach container2
```

コンテナのネットワーク・スタックを確認したら、２つのイーサネット・インターフェースが見えます。１つはデフォルトの bridge ネットワークであり、もう１つは isolated_nw ネットワークです。

```
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:03
          inet addr:172.17.0.3  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:9001  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)

eth1      Link encap:Ethernet  HWaddr 02:42:AC:15:00:02
          inet addr:172.25.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe19:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

isolated_nw はユーザが定義したネットワークであり、Docker 内部 DNS サーバがネットワーク上の他コンテナに対する適切な名前解決をします。 container2 の内部では、container3 に対して名前で ping できるでしょう。

```
/ # ping -w 4 container3
PING container3 (172.25.3.3): 56 data bytes
64 bytes from 172.25.3.3: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.3.3: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.25.3.3: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.3.3: seq=3 ttl=64 time=0.097 ms

--- container3 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms
```

ただし、デフォルトの bridge ネットワークを使う場合は、この名前解決機能を利用できません。 container2 と container1 は、どちらもデフォルトのブリッジ・ネットワークに接続しています。このデフォルトのネットワーク上では、Docker は自動サービス・ディスカバリをサポートしません。そのため、container1 に対して名前で ping をしても、 /etc/hosts ファイルに記述がない限り失敗するでしょう。

```
/ # ping -w 4 container1
ping: bad address 'container1'
```

container1 の IP アドレスであれば、次のように処理できます。

```
/ # ping -w 4 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.095 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.075 ms
64 bytes from 172.17.0.2: seq=2 ttl=64 time=0.072 ms
64 bytes from 172.17.0.2: seq=3 ttl=64 time=0.101 ms

--- 172.17.0.2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.072/0.085/0.101 ms
```

container1 と container2 を接続したい場合は、 docker run --link コマンドを使います。すると、２つのコンテナは IP アドレスだけでなく、名前でも相互に通信可能となります。

container2 からデタッチして離れるには、CTRL-p CTRL-q を実行します。

この例では、container2 は両方のネットワークに接続しているため、container1 と container3 の両方と通信できます。しかし、container3 と container1 は同じネットワーク上に存在していないため、お互いに通信できません。確認のため、container3 にアタッチし、container1 の IP アドレスに対して ping を試みましょう。

```
$ docker attach container3
/ # ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
^C
--- 172.17.0.2 ping statistics ---
10 packets transmitted, 0 packets received, 100% packet loss
```

コンテナをネットワークに接続するには、実行中でも停止中でも可能です。しかし、docker network inspect が

表示するのは、実行中のコンテナのみです。

ユーザ定義ネットワークでコンテナをリンク

先の例では、ユーザ定義ネットワーク isolated_nw において、container2 は自動的に container3 の名前解決が可能でした。しかし、デフォルトの bridge ネットワークでは自動的に名前解決が行われません。そのため、後方互換性のある**レガシーのリンク機能**を使い続ける必要が求められます。

レガシーのリンクは、デフォルト bridge ネットワーク上で４つの主な機能を提供します。

- 名前解決
- --link=コンテナ名:エイリアス の形式で、リンクしたコンテナの別名を指定
- コンテナの接続性を安全にする（ --icc=false で分離 ）
- 環境変数の挿入

上の４つの機能を、例で使ったデフォルトではない isolated_nw のようなユーザ定義ネットワークと比較します。docker network では追加設定を行わないものとします。

DNS を使い自動的に名前解決
ネットワーク内のコンテナに対して、安全に隔離された環境を自動的に
複数のネットワークを動的に装着・取り外しできる能力
リンクしているコンテナに対しては --link オプションでエイリアス名を指定

先ほどの例で説明を続けます。isolated_nw に別のコンテナ container4 を作成しましょう。この時、 --link オプションを付ければ、同一ネットワーク上の他コンテナが名前解決に使える別名（エイリアス）を指定できます。

```
$ docker run --net=isolated_nw -itd --name=container4 --link container5:c5 busybox
01b5df970834b77a9eadbaff39051f237957bd35c4c56f11193e0594cfd5117c
```

--link の助けにより、container4 が container5 に接続するために、c5 という別名でも接続できます。

container4 の作成時、リンクしようとする container5 という名前のコンテナは、まだ作成していないのに注意してください。これが、デフォルト bridge におけるレガシーのリンク機能と、ユーザ定義ネットワークにおける新しいリンク機能とで異なる挙動の１つです。レガシーのリンクは静的（固定）です。コンテナに対するエイリアス名は固定であり、リンク対象のコンテナ再起動は許容されません。一方のユーザ定義ネットワークにおける新しいリンク機能であれば、動的な性質を持っています。リンク対象のコンテナ再起動は許容されますし、IP　アドレスの変更もできます。

それでは container4 を c4 としてリンクする container5 という名前の別コンテナを起動しましょう。

```
$ docker run --net=isolated_nw -itd --name=container5 --link container4:c4 busybox
72eccf2208336f31e9e33ba327734125af00d1e1d2657878e2ee8154fbb23c7a
```

予想通り、container4 は container5 に対して接続できるのは、コンテナ名とエイリアス c5 の両方です。そして、container5 は container4 に対しても、コンテナ名とエイリアス c4 で接続できます。

```
$ docker attach container4
/ # ping -w 4 c5
PING c5 (172.25.0.5): 56 data bytes
64 bytes from 172.25.0.5: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.0.5: seq=1 ttl=64 time=0.080 ms
```

```
64 bytes from 172.25.0.5: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.0.5: seq=3 ttl=64 time=0.097 ms

--- c5 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms

/ # ping -w 4 container5
PING container5 (172.25.0.5): 56 data bytes
64 bytes from 172.25.0.5: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.0.5: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.25.0.5: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.0.5: seq=3 ttl=64 time=0.097 ms

--- container5 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms

$ docker attach container5
/ # ping -w 4 c4
PING c4 (172.25.0.4): 56 data bytes
64 bytes from 172.25.0.4: seq=0 ttl=64 time=0.065 ms
64 bytes from 172.25.0.4: seq=1 ttl=64 time=0.070 ms
64 bytes from 172.25.0.4: seq=2 ttl=64 time=0.067 ms
64 bytes from 172.25.0.4: seq=3 ttl=64 time=0.082 ms

--- c4 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.065/0.070/0.082 ms

/ # ping -w 4 container4
PING container4 (172.25.0.4): 56 data bytes
64 bytes from 172.25.0.4: seq=0 ttl=64 time=0.065 ms
64 bytes from 172.25.0.4: seq=1 ttl=64 time=0.070 ms
64 bytes from 172.25.0.4: seq=2 ttl=64 time=0.067 ms
64 bytes from 172.25.0.4: seq=3 ttl=64 time=0.082 ms

--- container4 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.065/0.070/0.082 ms
```

レガシーのリンク機能と新しいリンクのエイリアスは、コンテナに対してエイリアス名で接続するという意味では似ています。しかし、レガシーのリンクはコンテナに `--link` を指定した範囲でしか機能しません。

加えて、重要な注意点があります。コンテナが複数のネットワークに所属している場合、リンクのエイリアス（別名）が有効な範囲は、所属するネットワーク全体に適用されます。そのため、別のネットワークでは異なるエイリアスとしてリンクされる場合があります。

先ほどの例を進めます。 `local_alias` という別のネットワークを作成しましょう。

```
$ docker network create -d bridge --subnet 172.26.0.0/24 local_alias
76b7dc932e037589e6553f59f76008e5b76fa069638cd39776b890607f567aaa
```

`container4` と `container5` を新しい `local_aliases` ネットワークに接続します。

```
$ docker network connect --link container5:foo local_alias container4
```

```
$ docker network connect --link container4:bar local_alias conta

$ docker attach container4

/ # ping -w 4 foo
PING foo (172.26.0.3): 56 data bytes
64 bytes from 172.26.0.3: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.26.0.3: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.26.0.3: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.26.0.3: seq=3 ttl=64 time=0.097 ms

--- foo ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms

/ # ping -w 4 c5
PING c5 (172.25.0.5): 56 data bytes
64 bytes from 172.25.0.5: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.0.5: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.25.0.5: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.0.5: seq=3 ttl=64 time=0.097 ms

--- c5 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms
```

異なったネットワーク上でも ping が成功するのに注目してください。このセクションの結論を導くために、container5 を isolated_nw から切り離し、その結果を観察しましょう。

```
$ docker network disconnect isolated_nw container5

$ docker attach container4

/ # ping -w 4 c5
ping: bad address 'c5'

/ # ping -w 4 foo
PING foo (172.26.0.3): 56 data bytes
64 bytes from 172.26.0.3: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.26.0.3: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.26.0.3: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.26.0.3: seq=3 ttl=64 time=0.097 ms

--- foo ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms
```

結論として、ユーザ定義ネットワークにおける新しいリンク機能は、 従来のリンク機能が抱えていた問題を解決しているため、あらゆる面でレガシーのリンク機能より優位と言えます。

レガシーのリンク 機能と比較する場合、失われた機能の 1 つとして環境変数の挿入を注目すべきです。環境変数の挿入は非常に便利なものです。しかし、静的な性質であり、コンテナが開始する時に必ず挿入する必要がありました。環境変数を挿入できなかったのは、docker network との互換性を保つためです。これはネットワークにコンテナを動的に接続／切断する手法であり、環境変数の挿入は、実行中のコンテナに対して影響を与えてしまうか

らです。

ネットワーク範囲のエイリアス

リンク機能はコンテナ内におけるプライベートな名前解決を提供します。**ネットワーク範囲のエイリアス**（**network-scoped alias**）とは、特定のネットワークの範囲内でコンテナのエイリアス名を有効にします。

先ほどの例を続けます。isolated_nw でネットワーク・エイリアスを有効にした別のコンテナを起動します。

```
$ docker run --net=isolated_nw -itd --name=container6 --net-alias app busybox
8ebe6767c1e0361f27433090060b33200aac054a68476c3be87ef4005eb1df17

$ docker attach container4
/ # ping -w 4 app
PING app (172.25.0.6): 56 data bytes
64 bytes from 172.25.0.6: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.0.6: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=3 ttl=64 time=0.097 ms

--- app ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms

/ # ping -w 4 container6
PING container5 (172.25.0.6): 56 data bytes
64 bytes from 172.25.0.6: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.0.6: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=3 ttl=64 time=0.097 ms

--- container6 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms
```

container6 を local_alias ネットワークに接続しますが、異なったネットワーク範囲エイリアスを指定します。

```
$ docker network connect --alias scoped-app local_alias container6
```

この例における container6 は、isolated_nw では app とエイリアス名が指定されており、local_alias では scoped-app とエイリアス名が指定されています。

container4（両方のネットワークに接続）と container5（isolated_nw のみ接続 ）から接続できるか確認しましょう。

```
$ docker attach container4

/ # ping -w 4 scoped-app
PING foo (172.26.0.5): 56 data bytes
64 bytes from 172.26.0.5: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.26.0.5: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.26.0.5: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.26.0.5: seq=3 ttl=64 time=0.097 ms
```

```
--- foo ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms

$ docker attach container5

/ # ping -w 4 scoped-app
ping: bad address 'scoped-app'
```

ご覧の通り、ネットワーク範囲のエイリアスとは、ネットワークをエイリアスとしてアクセス可能に定義した範囲内のコンテナのみです。

この機能に加え、同一ネットワーク内であれば、複数のコンテナが同じネットワーク範囲としてのエイリアス名を共有できます。例えば isolated_nw に container7 を container6 と同じエイリアスで起動しましょう。

```
$ docker run --net=isolated_nw -itd --name=container7 --net-alias app busybox
3138c678c123b8799f4c7cc6a0cecc595acbdfa8bf81f621834103cd4f504554
```

複数のコンテナが同じエイリアス名を共有する時、エイリアスの名前解決はコンテナのいずれかで行います（通常は初めてエイリアス指定をしたコンテナです)。コンテナが停止してエイリアスが無効になるか、ネットワークから切断すれば、次のコンテナが名前解決のエイリアスに使われます。

container4 から app エイリアスに ping をした後、container6 を停止します。その後、app に対する名前解決が container7 になるのを確認しましょう。

```
$ docker attach container4
/ # ping -w 4 app
PING app (172.25.0.6): 56 data bytes
64 bytes from 172.25.0.6: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.0.6: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=3 ttl=64 time=0.097 ms

--- app ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms

$ docker stop container6

$ docker attach container4
/ # ping -w 4 app
PING app (172.25.0.7): 56 data bytes
64 bytes from 172.25.0.7: seq=0 ttl=64 time=0.095 ms
64 bytes from 172.25.0.7: seq=1 ttl=64 time=0.075 ms
64 bytes from 172.25.0.7: seq=2 ttl=64 time=0.072 ms
64 bytes from 172.25.0.7: seq=3 ttl=64 time=0.101 ms

--- app ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.072/0.085/0.101 ms
```

コンテナの切断

コンテナをネットワークから切断するには docker network disconnect（ディスコネクト）コマンドを使います。

```
$ docker network disconnect isolated_nw container2

docker inspect --format='{{json .NetworkSettings.Networks}}'  container2 | python -m json.tool
{
    "bridge": {
        "EndpointID": "9e4575f7f61c0f9d69317b7a4b92eefc133347836dd83ef65deffa16b9985dc0",
        "Gateway": "172.17.0.1",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "IPAddress": "172.17.0.3",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "MacAddress": "02:42:ac:11:00:03"
    }
}


$ docker network inspect isolated_nw
[
    {
        "Name": "isolated_nw",
        "Id": "06a62f1c73c4e3107c0f555b7a5f163309827bfbbf999840166065a8f35455a8",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.21.0.0/16",
                    "Gateway": "172.21.0.1/16"
                }
            ]
        },
        "Containers": {
            "467a7863c3f0277ef8e661b38427737f28099b61fa55622d6c30fb288d88c551": {
                "Name": "container3",
                "EndpointID": "dffc7ec2915af58cc827d995e6ebdc897342be0420123277103c40ae35579103",
                "MacAddress": "02:42:ac:19:03:03",
                "IPv4Address": "172.25.3.3/16",
                "IPv6Address": ""
            }
        },
        "Options": {}
    }
]
```

コンテナをネットワークから切断したら、対象ネットワーク上で接続していたコンテナと通信できなくなります。この例では、container2 は isolated_nw ネットワーク上の container3 とは通信できなくなります。

```
$ docker attach container2

/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:03
          inet addr:172.17.0.3  Bcast:0.0.0.0  Mask:255.255.0.0
```

```
          inet6 addr: fe80::42:acff:fe11:3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:9001  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ # ping container3
PING container3 (172.25.3.3): 56 data bytes
^C
--- container3 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
```

container2 は、ブリッジ・ネットワークに対する接続性をまだ維持しています。

```
/ # ping container1
PING container1 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.119 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.174 ms
^C
--- container1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.119/0.146/0.174 ms
/ #
```

複数ホストのネットワークにおいて、不意に docker デーモンの再起動が発生するシナリオを考えます。デーモンは接続していたエンドポイントとの接続性を解消していないものとします。エンドポイントでは、新しいコンテナがかつてと同じ名前で接続しようとしても container already connected to network（コンテナは既にネットワークに接続している）とエラーが出るかもしれません。エンドポイントの認識が古いのを解消するには、まず最初にコンテナを削除し、エンドポイントのネットワークから強制的に切断します（docker network disconnect -f）。エンドポイントがクリーンアップされれば、コンテナはネットワークに接続できるようになります。

```
$ docker run -d --name redis_db --net multihost redis
ERROR: Cannot start container bc0b19c089978f7845633027aa3435624ca3d12dd4f4f764b61eac4c0610f32e:
container already connected to network multihost

$ docker rm -f redis_db
$ docker network disconnect -f multihost redis_db

$ docker run -d --name redis_db --net multihost redis
7d986da974aeea5e9f7aca7e510bdb216d58682faa83a9040c2f2adc0544795a
```

ネットワークの削除

ネットワーク上の全てのコンテナが停止するか切断したら、ネットワークを削除できます。

```
$ docker network disconnect isolated_nw container3

docker network inspect isolated_nw
[
    {
        "Name": "isolated_nw",
        "Id": "06a62f1c73c4e3107c0f555b7a5f163309827bfbbf999840166065a8f35455a8",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.21.0.0/16",
                    "Gateway": "172.21.0.1/16"
                }
            ]
        },
        "Containers": {},
        "Options": {}
    }
]

$ docker network rm isolated_nw
```

全てのネットワーク情報を確認したら、isolated_nw が削除されています。

```
$ docker network ls
NETWORK ID          NAME                DRIVER
72314fa53006        host                host
f7ab26d71dbd        bridge              bridge
0f32e83e61ac        none                null
```

5.3.4 マルチホスト・ネットワーク機能を始める

このセクションでは、**マルチホスト・ネットワーク機能**（**multi-host networking**）の基本的な例について説明します。独創的な overlay ネットワーク・ドライバにより、Docker エンジンはマルチホスト・ネットワーキングをサポートしました。bridge ネットワークとは違い、オーバレイ・ネットワークを作成する前に、いくつかの事前準備が必要です。準備とは次のようなものです。

- kernel バージョン 3.16 以上のホスト。
- キーバリュー・ストアに対するアクセス。エンジンがサポートするキーバリュー・ストアは、Consul、Etcd、Zookeeper（分散ストア）。
- ホストのクラスタが、キーバリュー・ストアに接続する。
- Swarm の各ホスト上で動作する Docker Engine の daemon に、適切な設定を行う。
- クラスタ上のホストはユニークなホスト名を持つ必要がある。これは、キーバリュー・ストアがクラスタのメンバをホスト名で識別するため。

Docker マルチホスト・ネットワーク機能を使うために、Docker Machine と Docker Swarm の使用は強制ではありません。しかし、この例では Machine と Swarm を使用して説明します。Machine を使って**キーバリュー・ストア**用のサーバと、ホストのクラスタを作成します。この例では **Swarm クラスタ**を作成します。

動作条件

始める前に、自分のネットワーク上のシステムに、最新バージョンの Docker エンジン と Docker Machine をインストールしているか確認してください。この例では VirtualBox を扱います。Mac や Windows で Docker Toolbox を使ってインストールした場合は、いずれも最新バージョンをインストールしています。

もし準備ができていなければ、Docker Engine と Docker Machine を最新バージョンに更新してください。

ステップ１：キーバリュー・ストアのセットアップ

オーバレイ・ネットワークはキーバリュー・ストアが必要です。キーバリュー・ストアにネットワーク情報に関する情報を保管します。情報とは、ディスカバリ、ネットワーク、エンドポイント、IP アドレスなどです。Docker はキーバリュー・ストアとして Consul、Etcd、ZooKeeper（分散ストア）をサポートしています。今回の例では Consul を使います。

1. 必要な Docker Engine、Docker Machine、VirtualBox ソフトウェアを準備したシステムにログインします。

2. `mh-keystore` という名前の VirtualBox マシンを作成します。

```
$ docker-machine create -d virtualbox mh-keystore
```

新しいマシンを作成したら、Docker Engine をホスト上に追加する処理が行われます。つまりこれは、Consul を手動でインストールするのではなく、 Docker Hub 上の consul イメージ[*1] を使用するインスタンスの作成を意味します。インストールは次の手順で行います。

3. ローカルの環境変数を `mh-keystore` マシンに設定します。

```
$ eval "$(docker-machine env mh-keystore)"
```

4. `mh-keystore` マシン上で `progrium/consul` コンテナを起動します。

```
$ docker run -d \
    -p "8500:8500" \
    -h "consul" \
    progrium/consul -server -bootstrap
```

実行中の `mh-keystore` マシン上で 、クライアントは `progrium/consul` イメージを起動します。このサーバは `consul` と呼ばれており、ポート `8500` を開きます。

5. `docker ps` コマンドを実行したら、 `consul` コンテナが表示されます。

*1 https://hub.docker.com/r/progrium/consul/

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
  PORTS                                                                            NAMES
4d51392253b3        progrium/consul     "/bin/start -server -"   25 minutes ago      Up 25 minutes
 53/tcp, 53/udp, 8300-8302/tcp, 0.0.0.0:8500->8500/tcp, 8400/tcp, 8301-8302/udp   admiring_panini
```

ターミナルを開いたまま、次のステップに移ります。

ステップ２：Swarm クラスタの作成

このステップは docker-machine を使い、ネットワーク上にホストを準備します。この時点では実際にネットワークを作成する必用はありません。VirtualBox 内に複数のマシンを作成します。マシンの 1 つを **Swarm マスタ**として動かします（これは一番始めに作成するマシンです)。各々のホストを作成した後、マシン上のエンジンで overlay ネットワーク・ドライバに対応するために必要なオプション設定を追加します。

1. Swarm マスタを作成します。

```
$ docker-machine create \
-d virtualbox \
--swarm --swarm-master \
--swarm-discovery="consul://$(docker-machine ip mh-keystore):8500" \
--engine-opt="cluster-store=consul://$(docker-machine ip mh-keystore):8500" \
--engine-opt="cluster-advertise=eth1:2376" \
mhs-demo0
```

作成時、Docker Engine のデーモンに対して--cluster-store（クラスタストア）オプションを与えます。このオプションは、エンジンに対して overlay ネットワークのキーバリュー・ストアを伝えます。bash 変数展開 $(docker-machine ip mh-keystore) は、「ステップ 1」で作成した Consul サーバの IP アドレスを割り当てます。 --cluster-advertise（クラスタアドバタイズ）オプションは、ネットワーク上のマシンに対して公表（advertise）するものです。

2. Swarm クラスタに追加する他のホストを作成します。

```
$ docker-machine create -d virtualbox \
    --swarm \
    --swarm-discovery="consul://$(docker-machine ip mh-keystore):8500" \
    --engine-opt="cluster-store=consul://$(docker-machine ip mh-keystore):8500" \
    --engine-opt="cluster-advertise=eth1:2376" \
  mhs-demo1
```

3. マシン一覧から、全てのマシンが起動・実行中なのが確認します。

```
$ docker-machine ls
NAME         ACTIVE   DRIVER       STATE     URL                         SWARM
default      -        virtualbox   Running   tcp://192.168.99.100:2376
mh-keystore  *        virtualbox   Running   tcp://192.168.99.103:2376
mhs-demo0    -        virtualbox   Running   tcp://192.168.99.104:2376   mhs-demo0 (master)
mhs-demo1    -        virtualbox   Running   tcp://192.168.99.105:2376   mhs-demo0
```

この時点で、ネットワーク上に複数のホストが起動します。これらのホストを使って、マルチホスト・ネットワークを作成する準備が整いました。

ターミナルを開いたまま、次の手順に進みます。

ステップ３：オーバレイ・ネットワークの作成

オーバレイ・ネットワークを作成するには、次のようにします。

1. docker 環境変数を Swarm マスタのものにします。

```
$ eval $(docker-machine env --swarm mhs-demo0)
```

docker-machine に --swarm フラグを使えば、 docker コマンドは Swarm 情報のみ表示します。

2. docker info コマンドで Swarm クラスタの情報を表示します。

```
$ docker info
Containers: 3
Images: 2
Role: primary
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 2
mhs-demo0: 192.168.99.104:2376
 └ Containers: 2
 └ Reserved CPUs: 0 / 1
 └ Reserved Memory: 0 B / 1.021 GiB
 └ Labels: executiondriver=native-0.2, kernelversion=4.1.10-boot2docker, operatingsystem=Boot2Docker
1.9.0-rc1 (TCL 6.4); master : 4187d2c - Wed Oct 14 14:00:28 UTC 2015, provider=virtualbox,
storagedriver=aufs
mhs-demo1: 192.168.99.105:2376
 └ Containers: 1
 └ Reserved CPUs: 0 / 1
 └ Reserved Memory: 0 B / 1.021 GiB
 └ Labels: executiondriver=native-0.2, kernelversion=4.1.10-boot2docker, operatingsystem=Boot2Docker
1.9.0-rc1 (TCL 6.4); master : 4187d2c - Wed Oct 14 14:00:28 UTC 2015, provider=virtualbox,
storagedriver=aufs
CPUs: 2
Total Memory: 2.043 GiB
Name: 30438ece0915
```

この情報から、３つのコンテナが動作中で、マスタ上には２つのイメージがあるのが分かります。

3. overlay ネットワークを作成します。

```
$ docker network create --driver overlay --subnet=10.0.9.0/24 my-net
```

クラスタ上のどこかのホストで、ネットワークを作成する必要があります。この例では、Swarm マスタを使いますが、クラスタ上のホストであれば、どこでも簡単にできます。

ネットワークの作成時は --subnet オプションの指定を強く推奨します。 --subnet を指定しなければ、docker デーモンはネットワークに対してサブネットを自動的に割り当てます。その時、Docker が管理していない基盤上の別サブネットと重複する可能性が有り得ます。このような重複により、コンテナがネットワーク接続時に問題や障害を引き起こします。

4. ネットワークの状態を確認します。

```
$ docker network ls
NETWORK ID          NAME                DRIVER
412c2496d0eb        mhs-demo1/host      host
dd51763e6dd2        mhs-demo0/bridge    bridge
6b07d0be843f        my-net              overlay
b4234109bd9b        mhs-demo0/none      null
1aeead6dd890        mhs-demo0/host      host
d0bb78cbe7bd        mhs-demo1/bridge    bridge
1c0eb8f69ebb        mhs-demo1/none      null
```

Swarm マスタ環境にいるため、このように Swarm エージェントが動作している全てのネットワークを表示します。各 NETWORK　ID はユニークなことに注意します。各エンジンのデフォルト・ネットワークとオーバレイ・ネットワークが 1 つ表示されます

各 Swarm エージェントに切り替えて、ネットワークの一覧を見てみます。

```
$ eval $(docker-machine env mhs-demo0)
$ docker network ls
NETWORK ID          NAME                DRIVER
6b07d0be843f        my-net              overlay
dd51763e6dd2        bridge              bridge
b4234109bd9b        none                null
1aeead6dd890        host                host
$ eval $(docker-machine env mhs-demo1)
$ docker network ls
NETWORK ID          NAME                DRIVER
d0bb78cbe7bd        bridge              bridge
1c0eb8f69ebb        none                null
412c2496d0eb        host                host
6b07d0be843f        my-net              overlay
```

どちらのエージェントも、ID が 6b07d0be843f の my-net ネットワークを持っていると表示しています。これでマルチホスト・コンテナ・ネットワークが動作しました！

ステップ４：ネットワークでアプリケーションの実行

ネットワークを作成した後は、あらゆるホスト上で、自動的にこのネットワークの一部としてコンテナを開始できます。

1. Swarm マスタの環境変数を表示します。

```
$ eval $(docker-machine env --swarm mhs-demo0)
```

2. mhs-demo0 上に Ｎｇｉｎｘ（エンジンエックス）サーバを開始します。

```
$ docker run -itd --name=web --net=my-net --env="constraint:node==mhs-demo0" nginx
```

3. mhs-demo1 インスタンス上で BusyBox（ビジーボックス） インスタンスを実行し、Nginx サーバのホームページを表示します。

```
$ docker run -it --rm --net=my-net --env="constraint:node==mhs-demo1" busybox wget -O- http://web
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
ab2b8a86ca6c: Pull complete
2c5ac3f849df: Pull complete
Digest: sha256:5551dbdfc48d66734d0f01cafee0952cb6e8eeecd1e2492240bf2fd9640c2279
Status: Downloaded newer image for busybox:latest
Connecting to web (10.0.0.2:80)
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
}
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>


<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>


<p><em>Thank you for using nginx.</em></p>
</body>
</html>
-                    100% |********************************|   612   0:00:00 ETA
```

ステップ５：外部への疎通を確認

これまで見た通り、Docker 内部のオーバレイ・ネットワーク・ドライバによって、複数のホスト上（同じネットワークでなくとも）に存在するコンテナ間に、革新的な接続性をもたらします。更に、マルチホスト・ネットワークに接続するコンテナは、自動的に docker_gwbridge（ドッカーゲートウェイブリッジ） ネットワークに接続します。このネットワークはコンテナがクラスタの外部に対する疎通（接続性）をもたらします。

1. 環境変数を Swarm エージェントに切り替えます。

```
$ eval $(docker-machine env mhs-demo1)
```

2. ネットワーク一覧に docker_gwbridge ネットワークがあることを確認します。

```
$ docker network ls
NETWORK ID          NAME                DRIVER
6b07d0be843f        my-net              overlay
dd51763e6dd2        bridge              bridge
b4234109bd9b        none                null
1aeead6dd890        host                host
e1dbd5dff8be        docker_gwbridge     bridge
```

3. Swarm マスタでステップ 1 と 2 を繰り返します。

```
$ eval $(docker-machine env mhs-demo0)
$ docker network ls
NETWORK ID          NAME                DRIVER
6b07d0be843f        my-net              overlay
d0bb78cbe7bd        bridge              bridge
1c0eb8f69ebb        none                null
412c2496d0eb        host                host
97102a22e8d2        docker_gwbridge     bridge
```

4. Nginx コンテナのネットワーク・インターフェースを確認します。

```
$ docker exec web ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
inet6 ::1/128 scope host
    valid_lft forever preferred_lft forever
22: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
link/ether 02:42:0a:00:09:03 brd ff:ff:ff:ff:ff:ff
inet 10.0.9.3/24 scope global eth0
    valid_lft forever preferred_lft forever
inet6 fe80::42:aff:fe00:903/64 scope link
    valid_lft forever preferred_lft forever
24: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
inet 172.18.0.2/16 scope global eth1
    valid_lft forever preferred_lft forever
inet6 fe80::42:acff:fe12:2/64 scope link
    valid_lft forever preferred_lft forever
```

eth0 インターフェースは、コンテナが my-net オーバレイ・ネットワークに接続するインターフェースを表しています。eth1 インターフェースは、コンテナが docker_gwbridge ネットワークが接続するインターフェースを表します。

ステップ 6 ：Docker Compose との連係機能

Compose v2 フォーマットで導入された新しい機能を参照し、上記の Swarm クラスタを使ったマルチホスト・ネットワーク機能のシナリオをお試しください。

5.3.5 ユーザ定義ネットワーク用の内部 DNS サーバ

このセクションで扱う情報は、**内部 DNS サーバ（embedded DNS server）**をユーザ定義ネットワーク上で操作する方法です。ユーザ定義ネットワークに接続したコンテナは、DNS の名前解決の仕方がデフォルトの bridge ネットワークとは異なります。

後方互換性を維持するため、デフォルトの bridge ネットワークにおける DNS 設定方法は何も変わりません。デフォルトの bridge ネットワークにおける DNS 設定に関する詳しい情報は「コンテナの DNS を設定」のセクションをご覧ください。

Docker 1.10 では、docker デーモンに内部 DNS サーバを実装しました。これはコンテナ作成時の有効な**名前（name）**もしくは**ネット・エイリアス（net-alias）**または**リンク（link）の別名**を元にした**サービス・ディスカバリ**を提供します。Docker がコンテナ内で DNS 設定を管理する手法は、以降の Docker バージョンでは変わります。そのため、コンテナ内にある /etc/hosts と /etc/resolv.conf のようなファイルを考慮する必要はなくなり、これらのファイルはそのままにし、以下で挙げる Docker のオプションが指定できます。

複数のコンテナに対するオプションは、コンテナのドメインネーム・サービス（DNS）に影響を与えます。

- --name=コンテナ名 … --name で設定するコンテナ名は、ユーザ定義 Docker ネットワーク内でディスカバリ用に使われます。内部 DNS サーバは、このコンテナ名と（コンテナが接続するネットワーク上の）IP アドレスに対応し続けます。
- --net-alias=エイリアス名 … ユーザ定義ネットワーク内では、コンテナを見つける（ディスカバリする）ためには、この上にある --name に加えて 1 つまたは複数の --net-alias を指定できます（あるいは、docker network connect コマンドで --alias を使います）。内部 DNS サーバは、指定したユーザ定義ネットワーク内において、全てのコンテナ・エイリアス名（別名）と IP アドレスを対応し続けます。 docker network connect コマンドで --alias を使うことで、ネットワークごとに別々のエイリアス名が利用できます。
- --link=コンテナ名:エイリアス名 … このオプションをコンテナの run（実行）時に指定したら、内部 DNS は**エイリアス名**の追加エントリを指定します。これは**コンテナ名**を指定したコンテナの IP アドレスに対して、別の名前でアクセスできるようにします。内部 DNS では --link が指定されたら、 --link が指定されたコンテナの中から、唯一の結果のみ返します[*1]。これにより、内部のプロセスがコンテナの名前や IP アドレスを知らなくても、新しいコンテナに接続できるようにします。
- --dns=[IP アドレス...] … コンテナから名前解決のリクエストがあっても、内部 DNS サーバが名前解決できない時、DNS クエリを --dns オプションで指定した IP アドレスに転送します。 --dns の IP アドレスは内部 DNS サーバによって管理されるため、コンテナ内の /etc/resolv.conf ファイルを変更しません。
- --dns-search=ドメイン名... … コンテナ内部で使うホスト名にドメイン名が含まれていない時に、検索用に使うドメイン名を指定します[*2]。 --dns-search オプションは内部 DNS サーバによって管理されるため、コンテナ内の /etc/resolv.conf ファイルを変更しません。
- --dns-opt=オプション... … DNS リゾルバが使うオプションを設定します。これらオプションは内部 DNS サーバによって管理されるため、コンテナ内の /etc/resolv.conf ファイルを編集しません。利用可能なオプションについては、resolv.conf のドキュメントをご覧ください。

--dns=IP アドレス... 、 --dns-search=ドメイン名... 、 --dns-opt=オプション... の指定がなければ、Docker はホストマシン上（docker デーモンの実行環境）の /etc/resolv.conf を使います。この時、Docker デーモンは、ホスト上のオリジナル・ファイル上にある nameserver のエントリ、ここにある localhost の IP アドレス全てをフ

*1 訳者注：DNS ラウンドロビン方式の負荷分散に応用できます。

*2 訳者注：/etc/resolv.conf の search オプションと同じ機能です。

ィルタします[*1]。

フィルタリングが必要なのは、コンテナのネットワークから、ホスト上の localhost のアドレス全てに到達できるとは限らないためです。フィルタリング後は、コンテナ内の `/etc/resolv.conf` ファイルに `nameserver` のエントリが一切なくなります。そしてデーモンはコンテナの DNS 設定として、パブリックな Google DNS ネームサーバ（`8.8.8.8` と `8.8.4.4`）を追加します。デーモンで IPv6 が有効であれば、パブリックな Google の IPv6 DNS ネームサーバ（`2001:4860:4860::8888` と `2001:4860:4860::8844`）を追加します。

ホスト側のローカルホストにあるリゾルバにアクセスするには、コンテナ内から DNS サーバに到達可能になるように、ローカルホスト以外からも接続可能になるよう、リッスンする必要があります。

5.3.6 Docker デフォルトのブリッジ・ネットワーク

Docker ネットワーク機能の導入部では、自分自身で定義したネットワークを作成できました。Docker **デフォルトのブリッジ・ネットワーク**（**bridge network**）は、Docker Engine のインストール時に作成されたものです。このブリッジ・ネットワークは、単に **ブリッジ**（**bridge**）とも呼ばれます。以下のセクションでは、デフォルトの `bridge` ネットワークに関連する話題を扱います。

- コンテナ通信の理解
- レガシーのコンテナ・リンク機能
- コンテナのポートをホストに割り当て
- 自分でブリッジを作成
- コンテナの DNS を設定
- Docker0 ブリッジのカスタマイズ
- Docker と IPv6

コンテナ通信の理解

このセクションでは、Docker デフォルトのブリッジ・ネットワーク内部におけるコンテナ通信について説明します。このネットワークは `bridge` という名称のブリッジ・ネットワークであり、Docker インストール時に自動的に作成されます。

Docker ネットワーク機能を使えば、デフォルトのブリッジ・ネットワークに加え、自分で定義したネットワークも作成できます。

外の世界との通信

コンテナが世界と通信できるかどうかは、２つの要素が左右します。１つめの要素は、ホストマシンが IP パケットを転送できるかどうかです。２つめはホスト側の iptables（アイピーテーブルズ）が特定の接続を許可するかどうかです。

IP パケット転送（**packet forwarding**（パケット フォアワーディング））は、`ip_forward` システム・パラメータで管理します。このパラメータが 1 の時のみ、パケットは通信できます。通常、Docker サーバはデフォルトの設定のままでも `--ip-forward=true` であり、Docker はサーバの起動時に `ip_forward` を 1 にします。もし `--ip-forward=false` をセットし、システム・カーネルが有効な場合は、この `--ip-forward=false` オプションは無効です。カーネル設定の確認は、手動で行

*1 訳者注：無視します。

います。

```
$ sysctl net.ipv4.conf.all.forwarding
net.ipv4.conf.all.forwarding = 0
$ sysctl net.ipv4.conf.all.forwarding=1
$ sysctl net.ipv4.conf.all.forwarding
net.ipv4.conf.all.forwarding = 1
```

Docker を使う多くの環境で ip_forward の有効化が必要となるでしょう。コンテナと世界が通信できるようにするには、この設定が最低限必要だからです。また、複数のブリッジをセットアップする場合は、コンテナ間での通信にも必要となります。

デーモン起動時に --iptables=false を設定したら、Docker はシステム上の iptables ルールセットを一切変更しません。そうでなければ、Docker サーバは **DOCKER フィルタ・チェーン**の転送ルールを追加します。

Docker は DOCKER フィルタ・チェーンのために、既存のルールを削除・変更しません。そのため、ユーザが必要であれば、コンテナに対して更なるアクセス制限するといった、高度なルールも作成できます。

Docker のデフォルト転送ルールは、全ての外部ソースの IP アドレス対して許可しています。コンテナを特定の IP アドレスやネットワークに対してのみ接続したい場合には、DOCKER フィルタ・チェーンの一番上にネガティブ・ルールを追加します。例えば、コンテナが外部の IP アドレス 8.8.8.8 をソースとするもの**しか**許可しない場合には、次のようなルールを追加します。

```
$ iptables -I DOCKER -i ext_if ! -s 8.8.8.8 -j DROP
```

ext_if の場所は、インターフェースが提供するホスト側に接続できる名前[*1] です。

コンテナ間の通信

２つのコンテナが通信できるかどうかは、オペレーティング・システム・レベルでの２つの要素に左右されます。

- コンテナのネットワーク・インターフェースがネットワーク・トポロジに接続されていますか？ デフォルトの Docker は、全てのコンテナを docker0 ブリッジに接続するため、コンテナ間でのパケット通信が可能な経路を提供します。他の利用可能なトポロジに関するドキュメントについては、後述します。
- iptables は特定の接続を許可していますか？ Docker はデーモンの起動時に --iptables=false を設定したら、システム上の iptables に対する変更を一切行いません。そのかわり、Docker サーバは FORWARD チェーンにデフォルトのルールを追加する時、デフォルトの --icc=true であれば空の ACCEPT ポリシーを追加します。もし --icc=false であれば DROP ポリシーを設定します。

--icc=true のままにしておくか、あるいは --icc=false にすべきかという方針の検討には、iptables を他のコンテナやメインのホストから守るかどうかです。例えば、恣意的なポート探査やコンテナに対するアクセスは、問題を引き起こすかもしれません。

もし、より高い安全のために --icc=false を選択した場合は、コンテナが他のサービスと相互に通信するには、どのような設定が必要でしょうか。この答えが、 --link=コンテナ名_または_ID:エイリアス オプションです。これについては、以前のセクションでサービス名について言及しました。もし Docker デーモンが --icc=false と iptables=true のオプションを指定したら、docker run は --link= オプションの情報を参照し、他のコンテナが新しいコンテナの公開用ポートに接続できるよう iptables の ACCEPT ルールのペアを追加します。この公開用ポートとは、Dockerfile の EXPOSE 行で指定していたものです。

*1 インターフェース名などを指定します。

--link= で指定するコンテナ名の値は、Docker が自動的に割り当てる stupefied_pare のような名前ではなく、 docker run の実行時に --name= で名前を割り当てておく必要があります。ホスト名でなければ、Docker は --link= オプションの内容を理解できません。

Docker ホスト上で iptables コマンドを実行したら、FORWARD チェーンの場所で、デフォルトのポリシーが ACCEPT か DROP かを確認できます。

```
# もし--icc=false なら DROP ルールはどのようになるでしょうか：

$ sudo iptables -L -n
...
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
DROP       all  --  0.0.0.0/0            0.0.0.0/0
...

# --icc=false の下で --link= を指定したら、
# 特定のポートに対する ACCEPT ルールを優先し
# その他のパケットを DROP するポリシーを適用します。

$ sudo iptables -L -n
...
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
DROP       all  --  0.0.0.0/0            0.0.0.0/0

Chain DOCKER (1 references)
target     prot opt source               destination
ACCEPT     tcp  --  172.17.0.2           172.17.0.3           tcp spt:80
ACCEPT     tcp  --  172.17.0.3           172.17.0.2           tcp dpt:80
```

ホストを広範囲にわたって公開する iptables のルールは、各コンテナが持つ実際の IP アドレスを通して公開されますのでご注意ください。そのため、あるコンテナから別のコンテナに対する接続は、前者のコンテナ自身が持っている IP アドレスからの接続に見えるでしょう。

過去のコンテナ・リンク機能

このセクションで説明する過去（レガシー）のコンテナ・リンク機能に関する情報は、Docker のデフォルト・ブリッジ内でのみ扱えます。デフォルト・ブリッジとは bridge という名称のブリッジ・ネットワークであり、Docker をインストールすると自動的に作成されます。

Docker にネットワーク機能を導入するまでは、この Docker **リンク機能**によって、あるコンテナから別のコンテナに対してコンテナ間で相互の発見をし、安全に転送する情報を得られました。これから Docker ネットワーク機能を学ぶのであれば注意点があります。今もリンク機能を使いコンテナを作成できます。ただし、デフォルトのブリッジ・ネットワークとユーザ定義ネットワークでは、サポートされている機能が異なるのでご注意ください。

このセクションではネットワーク・ポートの接続と、それらをデフォルトのブリッジネットワーク上のコンテナ上でリンクする方法を簡単に扱います。

ネットワークのポート・マッピングで接続

以前のセクションでは、Python Flask アプリケーションを動かすコンテナを、次のように作成しました。

```
$ docker run -d -P training/webapp python app.py
```

コンテナは内部ネットワークと IP アドレスを持っています（以前のセクションで、docker inspect コマンドを実行してコンテナの IP アドレスを確認しました)。Docker は様々なネットワーク設定を持っています。Docker ネットワーク機能の詳細は「5.5　ネットワーク設定」のセクションをご覧ください。

コンテナの作成時に -P フラグを使えば、自動的にコンテナ内部のネットワーク・ポートを、ランダムなハイポート（Docker ホスト上のエフェメラル・ポート範囲内）に割り当てます。次は docker ps を実行時、コンテナ内のポート 5000 が、ホスト側の 49115 に接続していると分かります。

```
$ docker ps nostalgic_morse
CONTAINER ID  IMAGE                  COMMAND       CREATED        STATUS        PORTS
   NAMES
bc533791f3f5  training/webapp:latest  python app.py 5 seconds ago  Up 2 seconds  0.0.0.0:49155->5000/tcp
 nostalgic_morse
```

また、コンテナのポートを特定のポートに**マッピングする（割り当てる）**には、 -p フラグを使う方法も見てきました。ここでは、ホスト側のポート 80 に、コンテナのポート 5000 を割り当てています。

```
$ docker run -d -p 80:5000 training/webapp python app.py
```

そしてこの方法は、なぜ悪い考えなのでしょうか。それは、特定のコンテナが特定のポートを拘束するからです。
そうではなく、コンテナのポートを割り当てるには、デフォルトのエフェメラル・ポート範囲内を使うよりも、自分でホスト側のポート範囲を指定した方が望ましいでしょう。

```
$ docker run -d -p 8000-9000:5000 training/webapp python app.py
```

これはコンテナのポート 5000 を、ホスト側のポート 8000 〜 9000 の範囲において、利用可能なポートをランダムに割り当てます。
また、 -p フラグは他の目的のためにも設定できます。デフォルトの -p フラグは、ホスト側マシンの全てのインターフェースに対する特定のポートを使用します。ですが、特定のインターフェースの使用を明示することが可能です。例えば、localhost のみの指定は、次のようにします。

```
$ docker run -d -p 127.0.0.1:80:5000 training/webapp python app.py
```

これはコンテナ内のポート 5000 を、ホスト側マシン上の localhost か 127.0.0.1 インターフェース上のポート 80 に割り当てます。
あるいは、コンテナ内のポート 5000 を、ホスト側へ動的に割り当てますが、localhost だけ使いたい時は次のようにします。

```
$ docker run -d -p 127.0.0.1::5000 training/webapp python app.py
```

また、UDP ポートを割り当てたい場合は、最後に /udp を追加します。例えば、次のように実行します。

```
$ docker run -d -p 127.0.0.1:80:5000/udp training/webapp python app.py
```

また、便利な docker port ショートカットについても学びました。これは現在ポートが割り当てられている情報も含みます。これは、特定のポートに対する設定を確認するのにも便利です。例えば、ホストマシン上の localhost にコンテナのポートを割り当てている場合、 docker port を実行すると次のような出力を返します。

```
$ docker port nostalgic_morse 5000
127.0.0.1:49155
```

複数のポート設定は、-p フラグを複数指定します。

リンクしているシステムに接続

このセクションが扱うのはデフォルトのブリッジ・ネットワークにおけるレガシーのリンク機能です。ユーザ定義ネットワーク上での詳しい情報は、セクション「5.2.4 コンテナのネットワーク」をご覧ください。

Docker コンテナが他のコンテナに接続する方法は、ネットワーク・ポートの割り当て（mapping）だけではありません。Docker には**リンク・システム（linking system）**もあります。これは、複数のコンテナを一緒にリンクするもので、あるコンテナから別のコンテナに対する接続情報を送信します。コンテナをリンクしたら、ソース・コンテナに関する情報が、受信者側のコンテナに送られます。これにより、受信者側は送信元のコンテナを示す説明データを選ぶことができます。

名前付けの重要さ

Docker でリンク機能を使う時、コンテナ名に依存します。既に見てきたように、各コンテナを作成すると自動的に名前が作成されます。実際、このガイドでは nostalgic_morse という古い友人のような名前でした。コンテナ名は自分でも名付けられます。この名付けは２つの便利な機能を提供します。

- コンテナに名前を付けるのは、コンテナの名前を覚えておくためなど、特定の役割には便利です。例えば、ウェブ・アプリケーションのコンテナには web と名付けます。
- Docker で他のコンテナが参照できるようにするための、**リファレンス・ポイント（参照地点）**を提供します。例えば、web コンテナを db コンテナへリンクします。

コンテナ名を指定するには --name フラグを使います。例：

```
$ docker run -d -P --name web training/webapp python app.py
```

これは新しいコンテナを起動し、 --name フラグでコンテナ名を web とします。コンテナ名は docker ps コマンドで見られます。

```
$ docker ps -l
CONTAINER ID  IMAGE                    COMMAND        CREATED       STATUS       PORTS
 NAMES
aed84ee21bde  training/webapp:latest python app.py  12 hours ago  Up 2 seconds 0.0.0.0:49154->5000/tcp
web
```

あるいは docker inspect を使い、表示結果からコンテナ名の確認もできます。

コンテナ名はユニーク（一意）である必要があります。つまり、web と呼べるコンテナは 1 つだけです。コンテナ名を再利用したい場合は、同じ名前で新しいコンテナを作成する前に、古いコンテナの削除（docker rm を使用）が必要です。あるいは別の方法として、docker run コマンドの実行時に --rm フラグを指定します。これは、コンテナが停止したら、直ちにコンテナを削除するオプションです。

リンクを横断する通信

コンテナに対するリンクによりお互いのことを**発見（discover）**し、あるコンテナから別のコンテナに対して安全に転送する情報を得られます。リンクを設定したら、送信元コンテナから送信先コンテナに対する導線を作成します。リンクを作成するには、 --link フラグを使います。まず、新しいコンテナを作成します。今回はデータベースを含むコンテナを作成します。

```
$ docker run -d --name db training/postgres
```

これは PostgreSQL データベースを含む training/postgres イメージを使い、db という名称のコンテナを作成します。

次は、先ほどの手順で web コンテナを既に作成しているのであれば、リンク可能なコンテナに置き換えるため、削除する必要があります。

```
$ docker rm -f web
```

次は、db コンテナにリンクする新しい web コンテナを作成します。

```
$ docker run -d -P --name web --link db:db training/webapp python app.py
```

これは先ほど作成した db コンテナを新しい web コンテナにリンクするものです。 --link フラグは次のような形式です。

```
--link <名前 or id>:エイリアス
```

名前の場所はリンクしようとしているコンテナ名の場所であり、 **エイリアス**はリンク名の別名です。 --link フラグは、次のような形式もあります。

```
--link <名前 or id>
```

このケースではエイリアスはコンテナ名と一致しています。先ほどの例は、次のようにも書き換えられます。

```
$ docker run -d -P --name web --link db training/webapp python app.py
```

次は、 docker inspect でリンクしたコンテナを確認しましょう。

```
$ docker inspect -f "{{ .HostConfig.Links }}" web
[/db:/web/db]
```

これで web コンテナは db コンテナに web/db としてリンクされました。これを使い、db コンテナに対する接続情報を得られます。

コンテナに対するリンクとは、実際には何をしているのでしょうか？ これまで学んだように、リンクとは、送

信元コンテナが送信先コンテナに送るため、自分自身の情報を提供します。今回の例では、送信先は web であり、元になる**ソース・コンテナ** db に関する接続情報を入手できます。これにより、Docker はコンテナ間で安全なトンネルを作成します。つまり、db コンテナを開始する時に、 -P や -p フラグを使う必要がありません。これはリンク機能の大きな利点です。これは、元のコンテナのポートを公開する必要がありません。今回の例えは、PostgreSQL データベースをネットワークに接続する必要はありません。

Docker が元コンテナから送信先コンテナに接続情報を渡すには、２つの方法があります。

- 環境変数
- /etc/hosts ファイルの更新

環境変数

Docker はリンクするコンテナに対する様々な**環境変数**を作成します。Docker は --link パラメータで指定したコンテナを対象とする環境変数を、自動的に作成します。また、Docke は参照元とするコンテナの環境変数も作成します。これらの環境変数を使うには、次のようにします。

- ソース・コンテナの Dockerfile で ENV コマンドを使用
- ソース・コンテナの開始時に、docker run コマンドで -e 、 --env 、 --env-file オプションを使用

これらの環境変数は、ディスカバリのプログラム化を実現します。これはターゲットのコンテナ内の情報に、ソース・コンテナに関連する情報を含みます。

【警告】重要な理解が必要なのは、Docker がコンテナに関して作成する 全ての環境変数が、リンクされたあらゆるコンテナで利用できることです。これにより、機密事項を扱うデータをコンテナに保管する場合は、セキュリティに関する重大な影響を及ぼす場合があります。

Docker は --list パラメータで指定したターゲットコンテナごとに <エイリアス>_名前 環境変数を作成します。例えば、新しいコンテナ web がデータベース・コンテナ db とリンクするためには --link db:webdb を指定します。すると Docker は web コンテナ内で WEBDB_NAME=/web/webdb 環境変数を作成します。

また Docker は、ソース・コンテナが公開している各ポートの環境変数も定義します。各変数には、ユニークな接頭語を付けています。

<名前>_PORT_<ポート番号>_<プロトコル>

この接頭語の要素は、次の通りです。

エイリアスの <名前> を --link パラメータで指定している場合（例： webdb ）
公開している <ポート> 番号
TCP もしくは UDP の <プロトコル>

Docker はこれら接頭語の形式を、３つの異なる環境変数で使います。

- prefix_ADDR 変数は、URL 用の IP アドレスを含む。例：WEBDB_PORT_5432_TCP_ADDR=172.17.0.82
- prefix_PORT 変数は、URL 用のポート番号を含む。例：WEBDB_PORT_5432_TCP_PORT=5432
- prefix_PROTO 変数は URL 用のプロトコルを含む。例：WEBDB_PORT_5432_TCP_PROTO=tcp

もしコンテナが複数のポートを公開している場合は、それぞれのポートを定義する環境変数が作成されます。つまり、例えばコンテナが 4 つのポートを公開しているのであれば、Docker はポートごとに 3 つの環境変数を作成するため、合計 12 個の変数を作成します。

更に、Docker は <エイリアス>_ポート の環境変数も作成します。この変数にはソース・コンテナが 1 番めに公開しているポートの URL を含みます。「1 番め」のポートとは、公開しているポートのうち、最も低い番号です。例えば、 WEBDB_PORT=tcp://172.17.0.82:5432 のような変数が考えられます。もし、ポートが tcp と udp の両方を使っているのであれば、tcp のポートだけが指定されます。

最後に、ソース・コンテナ上の Docker に由来する環境変数は、ターゲット上でも環境変数として使えるように公開されます。Docker が作成した各環境変数 <エイリアス>_ENV_<名前> が、ターゲットのコンテナから参照できます。これら環境変数の値は、ソース・コンテナが起動した時の値を使います。

データベースの例に戻りましょう。env コマンドを実行したら、指定したコンテナの環境変数一覧を表示します。

```
$ docker run --rm --name web2 --link db:db training/webapp env
. . .
DB_NAME=/web2/db
DB_PORT=tcp://172.17.0.5:5432
DB_PORT_5432_TCP=tcp://172.17.0.5:5432
DB_PORT_5432_TCP_PROTO=tcp
DB_PORT_5432_TCP_PORT=5432
DB_PORT_5432_TCP_ADDR=172.17.0.5
. . .
```

このように、Docker は環境変数を作成しており、そこには元になったソース・コンテナに関する便利な情報を含みます。各変数にある接頭語 DB_ とは、先ほど指定したエイリアスから割り当てられています。もしエイリアスが db1 であれば、環境変数の接頭語は DB1_になります。これらの環境変数を使い、アプリケーションが db コンテナ上のデータベースに接続する設定も可能です。接続は安全かつプライベートなものですが、これはリンクされた web コンテナと db コンテナが通信できるようにするだけです。

Docker 環境変数に関する重要な注意

/etc/host ファイルのエントリとは違い、もし元になったコンテナが再起動しても、保管されている IP アドレスの情報は自動的に更新されません。リンクするコンテナの IP アドレスを名前解決するには、 /etc/hosts エントリの利用をお勧めします。

これらの環境変数が作成されるのは、コンテナの初期段階のみです。sshd のようなデーモンであれば、シェルへの接続が生じた時に確定します。

/etc/hosts ファイルの更新

環境変数について追記しますと、Docker は /etc/hosts ファイルに、元になったコンテナのエントリを追加します。ここでは web コンテナのエントリを見てみましょう。

```
$ docker run -t -i --rm --link db:webdb training/webapp /bin/bash
root@aed84ee21bde:/opt/webapp# cat /etc/hosts
172.17.0.7  aed84ee21bde
. . .
172.17.0.5  webdb 6e5cdeb2d300 db
```

関係あるホスト 2 つのエントリが見えます。1 行めのエントリは、web コンテナのものであり、コンテナ ID がホスト名として使われています。2 つめのエントリは db コンテナのものであり、IP アドレスの参照にエイリアスが使われています。エイリアスの指定に加えて、もし --link パラメータで指定したエイリアスがユニークであれ

ば、リンクされるコンテナのホスト名もまた /etc/hosts でコンテナの IP アドレスをリンクします。これでホスト上では、これらのエントリを通して ping できます。

```
root@aed84ee21bde:/opt/webapp# apt-get install -yqq inetutils-ping
root@aed84ee21bde:/opt/webapp# ping webdb
PING webdb (172.17.0.5): 48 data bytes
56 bytes from 172.17.0.5: icmp_seq=0 ttl=64 time=0.267 ms
56 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.250 ms
56 bytes from 172.17.0.5: icmp_seq=2 ttl=64 time=0.256 ms
```

この例で ping をインストールしているのは、コンテナの初期状態では入っていないためです。

これで、db コンテナに対して ping コマンドを実行する時は、hosts エントリにある 172.17.0.5 を名前解決して ping します。この hosts のエントリの設定を使えば、アプリケーションが db コンテナに接続する設定で使えます。

１つのソース・コンテナから、複数の送信先コンテナにリンクできます。例えば、複数の（異なった名前の）ウェブ・コンテナが、db コンテナに接続できます。

ソース・コンテナを再起動したら、リンクされたコンテナの /etc/hosts ファイルはソース・コンテナの IP アドレスを自動的に更新し、継続して通信できるようにします。

```
$ docker restart db
db
$ docker run -t -i --rm --link db:db training/webapp /bin/bash
root@aed84ee21bde:/opt/webapp# cat /etc/hosts
172.17.0.7  aed84ee21bde
. . .
172.17.0.9  db
```

ホスト上にコンテナのポートを割り当て

このセクションでは、 Docker のデフォルト・ブリッジ内にあるコンテナに対して、ポートを割り当てる方法を説明します。このネットワークは bridge という名称のブリッジ・ネットワークであり、Docker インストール時に自動的に作成されるものです。

Docker ネットワーク機能を使えば、デフォルトのブリッジ・ネットワークに加え、自分で定義したネットワークも作成できます。

デフォルトの Docker コンテナは外の世界と通信できます。しかし、外の世界からはコンテナに接続できません。外に対するそれぞれの接続は、ホストマシン自身が持つ IP アドレスから行われているように見えます。これはホストマシン上の **iptables マスカレーディング（masquerading）**機能によるルールであり、Docker サーバ起動時に自動的に作成されます。

```
$ sudo iptables -t nat -L -n
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16       0.0.0.0/0
...
```

Docker サーバが作成するマスカレード・ルールは、外の世界からコンテナに IP アドレスで接続できるようにします。

もしもコンテナが内側（incoming）への接続を受け付けたいのなら、 docker run の実行時に特別なオプション指定が必要です。

まずは、docker run で -P か --publish-all=true|false を指定します。これは全体的なオプションであり、イメージの Dockerfile 内にある EXPOSE 命令で各ポートを指定するか、あるいは、コマンドラインで --expose <ポート> フラグを使って指定します。いずれかを使い、ホスト上の**エフェメラル・ポート範囲内（ephemeral port range）**にあるポートに割り当てられます。以後、どのポートに割り当てられているか調べるには docker port コマンドを使います。エフェメラル・ポート範囲とは、ホスト側の Linux カーネル・パラメータの /proc/sys/net/ipv4/ip_local_port_range で指定されており、典型的な範囲は 32768 〜 61000 です。

割り当て（マッピング）を明示するには、 -p 指定 か --publish=指定 オプションを使います。これは、Docker サーバのどのポートを使うか明示するものであり、ポートの指定が無ければ、エフェメラル・ポート範囲から割り当てられます。この指定を使い、コンテナの任意のポートに割り当て可能です。

どちらにしろ、Docker がネットワーク・スタックでどのような処理を行っているのかは、NAT テーブルを例に垣間見えます。

```
# Docker で -P 転送を指定した時、NAT ルールがどうなるか

$ iptables -t nat -L -n
...
Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:49153 to:172.17.0.2:80

# Docker で -p 80:80 を指定した時、NAT ルールがどうなるか

Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:80 to:172.17.0.2:80
```

Docker はこれらコンテナ側のポートを 0.0.0.0、つまりワイルドカード IP アドレスで公開しているのが分かります。これはホストマシン上に到達可能な全ての IP アドレスが対象です。制限を加えたい場合や、ホストマシン上の特定の外部インターフェースを通してのみコンテナのサービスへ接続したい場合は、2 つの選択肢があります。1 つは docker run の実行時、 -p IP:ホスト側ポート:コンテナ側ポート か -p IP::ポート を指定し、特定の外部インターフェースをバインドする指定ができます。

あるいは、Docker のポート転送で常に特定の IP アドレスに割り当てたい場合には、システム全体の Docker サーバ設定ファイルを編集し、 --ip=IP_ADDRESS オプションを使います。編集内容が有効になるのは、Docker サーバの再起動後なのでご注意ください。

ヘアピン NAT を有効にすると（--userland-proxy=false）、コンテナのポート公開は、純粋な iptables のルールを通して処理され、特定のポートを結び付けようとする処理は、一切ありません。つまり、コンテナが使おうとしているポートを、Docker の他の何かのサービスが使おうとしても阻止できません。このような衝突があれば、Docker はコンテナに対する経路を優先するよう、iptabels のルールを書き換えます。

--userland-proxy パラメータは、デフォルトでは true（有効）であり、コンテナ内部とコンテナ外からの通信を可能にするユーザ領域を提供します。無効化しますと、Docker はどちらの通信に対しても MASQUERADE iptables

ルールを追加します。 net.ipv4.route_localnet カーネル・パラメータを使い、ホストマシンがローカルのコンテナが公開しているポートに対し、通常はループバック・アドレスを用いて通信します。どちらを選ぶかは、性能を根拠として選びます。

自分でブリッジを作成

このセクションでは、Docker のデフォルト・ブリッジを自分自身で構築したブリッジに置き換える方法を説明します。bridge という名称のブリッジ・ネットワークは、Docker インストール時に自動的に作成されるものです。

Docker ネットワーク機能 を使えば、デフォルト・ブリッジ・ネットワークに加え、自分で定義したネットワークも作成できます。

自分自身のブリッジをセットアップするには、Docker を起動する前に Docker に対して -b ブリッジ名 か --bridge=ブリッジ名 を使い、特定のブリッジを指定します。既に Docker を起動している場合は、デフォルトの docker0 がありますが、自分でもブリッジを作成できます。必要があれば、サービスを停止してインターフェースの削除も可能です。

```
# Docker を停止し、docker0 の削除

$ sudo service docker stop
$ sudo ip link set dev docker0 down
$ sudo brctl delbr docker0
$ sudo iptables -t nat -F POSTROUTING
```

それから、Docker サービスを開始する前に、自分自身のブリッジを作成し、必要な設定を行います。ここではシンプルながら十分なブリッジを作成します。これまでのセクションで用いてきた docker0 をカスタマイズするだけですが、技術を説明するには十分なものです。

```
# 自分自身でブリッジを作成

$ sudo brctl addbr bridge0
$ sudo ip addr add 192.168.5.1/24 dev bridge0
$ sudo ip link set dev bridge0 up

# ブリッジが起動し、実行中なことを確認

$ ip addr show bridge0
4: bridge0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP group default
    inet 192.168.5.1/24 scope global bridge0
       valid_lft forever preferred_lft forever

# Docker にこの情報を登録し、再起動（Ubuntu の場合）

$ echo 'DOCKER_OPTS="-b=bridge0"' >> /etc/default/docker
$ sudo service docker start

# 新しく外側への NAT マスカレードが作成されたことを確認

$ sudo iptables -t nat -L -n
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  192.168.5.0/24      0.0.0.0/0
```

この結果、Docker サーバの起動は成功し、コンテナに対して新しいブリッジが割り当てられているでしょう。ブリッジの設定を確認した後、コンテナを作成してみます。Docker は新しい IP アドレスの範囲を自動的に検出し、IP アドレスが割り当てられているのが分かるでしょう。

brctl show コマンドを使えば、コンテナの開始・停止時に Docker がブリッジを追加・削除してるのが分かります。そして、コンテナの中で ip addr と ip route を実行したら、IP アドレスがブリッジの IP アドレス範囲内にあることが分かります。そして他のインターネットへのデフォルト・ゲートウェイとして、Docker ホストの IP アドレスをブリッジするのに使われます。

コンテナの DNS を設定

このセクションでは Docker のデフォルト・ブリッジ内でコンテナの DNS を設定する方法を紹介します。ここでの bridge という名称のブリッジ・ネットワークは、Docker インストール時に自動的に作成されるものです。

Docker ネットワーク機能を使えば、デフォルト・ブリッジ・ネットワークに加え、自分でユーザ定義ネットワークも作成できます。ユーザ定義ネットワーク上で DNS 設定を行う詳細な情報は、Docker 内部 DNS のセクションをご覧ください。

Docker が各コンテナにホスト名や DNS 設定を渡すため、カスタム・イメージを使わずコンテナ内にホスト名を書くには、どうしたら良いでしょうか。コンテナ内では、 /etc 以下にある 3 つの重要なファイルを、仮想ファイルを使って新しい情報に上書きする手法を使います。これは実行中のコンテナ内で mount コマンドを実行すると分かります。

```
$$ mount
...
/dev/disk/by-uuid/1fec...ebdf on /etc/hostname type ext4 ...
/dev/disk/by-uuid/1fec...ebdf on /etc/hosts type ext4 ...
/dev/disk/by-uuid/1fec...ebdf on /etc/resolv.conf type ext4 ...
...
```

この処理により、Docker は巧妙に resolv.conf の最新版を維持します。この情報はホストマシンが新しい設定を DHCP 経由で受信したら、全てのコンテナに対して反映します。Docker がコンテナの中でこれらのファイルをどう管理するのか正確な情報を持つため、Docker のバージョンを更新しても変わりません。そのため、ファイルの内容に対して変更を加えるのではなく、以下のオプションを使います。

コンテナのドメイン名サービスに影響を与える 4 つのオプションがあります。

- -h ホスト名 か --hostname=ホスト名 ：コンテナ自身が知るホスト名を設定します。これは /etc/hostname に書かれます。 /etc/hosts にはコンテナ内のホスト IP アドレスが書かれ、その名前がコンテナ内の /bin/bash プロンプトで表示されます。しかし、ホスト名をコンテナの外から確認するのは大変です。docker ps でも見えませんし、他のコンテナの /etc/hosts からも見えません。
- --link=コンテナ名 か ID:エイリアス ： 実行しているコンテナに対して、新しいコンテナの /etc/hosts にエイリアスという別名のエントリを追加します。コンテナを識別する IP アドレスが指し示すのは、 コンテナ名か ID です。これにより、新しいコンテナ内のプロセスは、IP アドレスを知らなくても、ホスト名の エイリアス を使って接続できます。 --link オプションの詳細については、以降で扱います。Docker は異なる IP アドレスが割り当てられる可能性があり、リンクされたコンテナを再起動したら、Docker は対象となるコンテナの /etc/hosts のエイリアスエントリを更新します。
- --dns=IP アドレス... ：コンテナの /etc/resolv.conf ファイルの server 行に IP アドレスを追加します。コンテナ内のプロセスは、 /etc/host を突き合わせて一致するホスト名が存在しなければ、ここで指定した IP アドレスのポート 53 を名前解決用に使います。

- --dns-search=ドメイン名... ： コンテナ内でホスト名だけが単体で用いられた時、ここで指定したドメイン名に対する名前解決を行います。これはコンテナ内の /etc/resolv.conf の search 行で書かれているものです。コンテナのプロセスがホスト名 host に接続しようとする時、search ドメインに example.com が指定されていれば、DNS 機構は host だけでなく、host.example.com に対しても名前解決を行います。検索用ドメインを指定したくない場合は、 --dns-search=. を使います。
- --dns-opt=オプション ：コンテナ内の /etc/resolv.conf の options 行で DNS 解決のオプションを指定します。利用可能なオプションについては、resolv.conf のオプションをご覧ください。

DNS 設定に関して、オプション --dns=IP アドレス... 、 --dns-search=ドメイン名... 、 --dns-opt=オプション... の指定がなければ、Docker は各コンテナの /etc/resolv.conf をホストマシン上（docker デーモンが動作中）の /etc/resolv.conf のように作成します。コンテナの /etc/resolv.conf の作成時、デーモンはホスト側のオリジナル・ファイルに書かれている nameserver のエントリを、フィルタリングして書き出します。

フィルタリングが必要になるのは、ホスト上の全てのローカルアドレスがコンテナのネットワークから到達できない場合があるからです。フィルタリングにより、コンテナ内の /etc/resolv.conf ファイルに nameserver のエントリが残っていなければ、デーモンはパブリック Google DNS ネームサーバ (8.8.8.8 と 8.8.4.4) をコンテナの DNS 設定に用います。もしデーモン上で IPv6 を有効にしているのであれば、パブリック IPv6 Google DNS ネームサーバを割り当てます（2001:4860:4860::8888 と 2001:4860:4860::8844)。

ホスト側のローカルなリゾルバにアクセスする必要があるなら、コンテナ内から到達可能になるように、ホスト上にある DNS サービスがローカルホスト以外をリッスンするよう設定が必要です。

ホストマシン側の /etc/resolv.conf を変更すると何が起こるか気になるでしょう。 docker デーモンはホスト側の DNS 設定に対する変更を監視していますので、ファイルの変更を通知します。

ファイル変更の通知は Linux カーネルの inotify 機能に依存しています。現時点では overlay ファイルシステム・ドライバとは互換性がありません。そのため Docker デーモンが「overlay」を使う場合は、 /etc/resolv.conf の自動更新機能を使えません。

host ファイルの変更される時、resolv.conf を持っている全てのコンテナ上で、host ファイルは直ちに最新の host 設定に更新されます。host 設定変更時にコンテナが実行中であれば、コンテナを停止・再起動して反映する必要があります。コンテナ実行中は、resolv.conf の内容はそのまま維持されるためです。もし、コンテナがデフォルトの設定で開始される前に resolv.conf を編集している場合は、ファイルをコンテナ内で編集した内容はそのままで、置き換えられることはありません。もしオプション（--dns、--dns-search、--dns-port）がデフォルトのホスト設定に用いられている場合は、同様にホスト側の/etc/resolv.conf を更新しません。

/etc/resolv.conf の更新機能が実装されているのは、Docker 1.5.0 以降に作られたコンテナです。つまり、以前のコンテナはホスト側の resolv.conf の変更が発生しても検出できません。Docker 1.5 以降に作成されたコンテナのみ、上記の自動更新機能が使えます。

Docker0 ブリッジのカスタマイズ

このセクションでは Docker のデフォルト・ブリッジをどのようにカスタマイズするか説明します。bridge という名称のブリッジ・ネットワークは、Docker インストール時に自動的に作成されるものです。

Docker ネットワーク機能を使えば、デフォルトのブリッジ・ネットワークに加え、自分で定義したネットワークも作成できます。

デフォルトでは、Docker サーバはホスト・システム上の docker0 インターフェースを Linux カーネル内部のイ

ーサネット・ブリッジ（Ethernet bridge）として設定・作成します。1つのイーサネット・ネットワークとして振る舞い、パケットの送受信や、別の物理ないし仮想ネットワーク・インターフェースに転送します。

Docker は docker0 の IP アドレス、ネットマスク、IP 割り当て範囲（レンジ）を設定します。ホストマシンはブリッジに接続したコンテナに対して、パケットの送受信が可能です。そして、MTU (maximum transmission unit) の指定値や、インターフェースが扱えるパケット値の超過を指定する値は、Docker ホスト上のデフォルトでルーティングされるインターフェース値からコピーされます。これらのオプションはサーバ起動時に設定可能です。例えば、 --bip-CIDER は bridge0 ブリッジに対して特定の IP アドレスとネットマスクを指定、ここでは 192.168.1.5/24 のような通常の CIDR（サイダー）で指定します。

- --fixed-cidr=CIDR ： docker0 サブネットが使う IP 範囲を、172.167.1.0/28 のような標準的な CIDR 形式で指定します。この範囲は IPv4 で固定する（例：10.20.0.0/16）必要があり、ブリッジの IP 範囲（docker0 あるいは --bridge で指定）のサブセットである必要もあります。例えば --fixed-cidr=192.168.1.0/25 を指定したら、コンテナの IP アドレスは前半の 192.168.1.0/24 サブネットから割り当てられます。
- --mtu=バイト数 ： docker0 上の最大パケット長を上書きします。

1つまたは複数のコンテナを実行し、ホストマシン上で brctl コマンドを実行したら、出力の interfaces 列から、Docker は docker0 ブリッジに適切に接続しているのが分かります。次の例は、ホスト上で2つの異なったコンテナが接続しています。

```
# ブリッジ情報の表示

$ sudo brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.3a1d7362b4ee       no              veth65f9
                                                        vethdda6
```

brctl コマンドが Docker ホスト上にインストールされていなければ、Ubuntu であれば sudo apt-get install bridge-utils でインストール可能です。

最後に、docker0 イーサネット・ブリッジの設定は新しいコンテナを作成する度に行われます。Docker は docker run で新しいコンテナを実行時、ブリッジは毎回利用可能な範囲にある空き IP アドレスを探します。それから、コンテナの eth0 インターフェースにその IP アドレスとブリッジのネットマスクを設定します。Docker ホスト自身が IP アドレスをブリッジする設定の場合は、ブリッジにデフォルト・ゲートウェイが用いられ、各コンテナが他のインターネット環境と接続できるようになります。

```
# コンテナから見えるネットワーク

$ docker run -i -t --rm base /bin/bash

$$ ip addr show eth0
24: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 32:6f:e0:35:57:91 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::306f:e0ff:fe35:5791/64 scope link
       valid_lft forever preferred_lft forever

$$ ip route
default via 172.17.42.1 dev eth0
172.17.0.0/16 dev eth0  proto kernel  scope link  src 172.17.0.3
```

```
$$ exit
```

Docker はホスト側の ip_forward システム設定が 1 でなければ、Docker ホストはコンテナのパケットをインターネット側に転送できないのでご注意ください。

Docker と IPv6

このセクションでは Docker のデフォルト・ブリッジ上の IPv6 を説明します。bridge という名称のブリッジ・ネットワークは、Docker インストール時に自動的に作成されるものです。

IPv4 アドレス枯渇問題[*1] により、IFTF は IPv4 の後継規格 IPv6[*2]（インターネット・プロトコル・バージョン 6）を RFC 2460[*3] で策定しました。IPv4 および IPv6 の両プロトコルは、OSI 参照モデル[*4] のレイヤ 3 にあたります。

Docker の IPv6 機能

デフォルトでは、 Docker サーバはコンテナ・ネットワークを IPv4 のみ設定します。Docker デーモンに --ipv6 フラグを指定して実行したら、IPv4/IPv6 デュアルスタック・サポートが有効になります。Docker は bridge0 の IPv6 リンク・ローカルアドレス[*5] に fe80::1 をセットアップします。

デフォルトでは、コンテナはリンク・ローカル IPv6 アドレスのみ割り当てられます。グローバルにルーティング可能な IPv6 アドレスを割り当てるには、コンテナに対して割り当てる特定の IPv6 サブネットを指定します。IPv6 サブネットを設定するには、 --fixed-cidr-v6 パラメータを Docker デーモンの起動時に指定します。

```
docker daemon --ipv6 --fixed-cidr-v6="2001:db8:1::/64"
```

Docker コンテナ用のサブネットは、少なくとも /80 を持っている必要があります。この方法により、IPv6 アドレスはコンテナの MAC アドレスで終わることができ、NDP ネイバー・キャッシュの無効化問題を Docker のレイヤで発生しないようにします。

--fixed-cidr-v6 パラメータを Docker に設定したら、新しい経路のルーティング・テーブルを作成します。更に IPv6 ルーティング・テーブルも有効化します（有効化したくない場合は、Docker デーモン起動時に --ip-forward=false を指定します）。

```
$ ip -6 route add 2001:db8:1::/64 dev docker0
$ sysctl net.ipv6.conf.default.forwarding=1
$ sysctl net.ipv6.conf.all.forwarding=1
```

サブネット 2001:db8:1::/64 に対する全てのトラフィックは、docker0 インターフェースを通る経路になります。

IPv6 転送（IPv6 forwarding）は既存の IPv6 設定に干渉する場合があり、注意が必要です。ホスト・インターフェースが IPv6 設定を取得するために、**ルータ・アドバタイズメント（Router Advertisement）**を使っているのであれば、accept_ra を 2 に設定すべきです。そうしなければ、IPv6 転送を有効化した結果、ルータ・アドバタイ

---

* 1 https://ja.wikipedia.org/wiki/IP%E3%82%A2%E3%83%89%E3%83%AC%E3%82%B9%E6%9E%AF%E6%B8%87%E5%95%8F%E9%A1%8C

*2 https://ja.wikipedia.org/wiki/IPv6

*3 https://www.ietf.org/rfc/rfc2460.txt

* 4 https://ja.wikipedia.org/wiki/OSI%E5%8F%82%E7%85%A7%E3%83%A2%E3%83%87%E3%83%AB

*5 http://en.wikipedia.org/wiki/Link-local_address

ズメントを拒否します。例えば、eth0 を経由してルータ・アドバタイズメントを使いたい場合は、次のように設定すべきです。

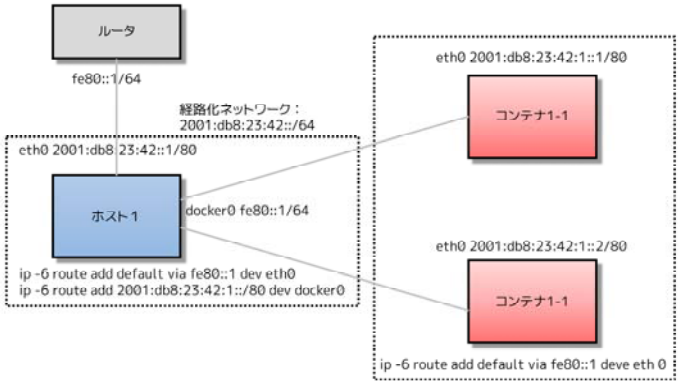
```
$ sysctl net.ipv6.conf.eth0.accept_ra=2
```

それぞれの新しいコンテナは、定義されたサブネットから IPv6 アドレスを取得します。更に、デフォルト経路（default route）がコンテナ内の eth0 に追加されます。これはデーモンのオプションで --default-gateway-v6 を指定しました。指定がなければ、fe80::1 経由になります。

```
docker run -it ubuntu bash -c "ip -6 addr show dev eth0; ip -6 route show"

15: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500
   inet6 2001:db8:1:0:0:242:ac11:3/64 scope global
      valid_lft forever preferred_lft forever
   inet6 fe80::42:acff:fe11:3/64 scope link
      valid_lft forever preferred_lft forever

2001:db8:1::/64 dev eth0  proto kernel  metric 256
fe80::/64 dev eth0  proto kernel  metric 256
default via fe80::1 dev eth0  metric 1024
```

この例では、Docker コンテナはネットワーク・サフィックス /64 で割り当てられた（ここでは fe80::42:acff:fe11:3/64 ）リンク・ローカル・アドレスと、グローバルな経路を持つ IPv6 アドレス（ここでは、2001:db8:1:0:0:242:ac11:3/64 ）を持ちます。コンテナは、リンク・ローカル・ゲートウェイ eth0 の fe80::1 を使い、2001:db8:1::/6 ネットワークの外と通信します。

サーバや仮想マシンは /64 IPv4 サブネットを割り当てられます（例：2001:db8:23:42::/64)。今回の例では、ホスト上の他のアプリケーションとの分離に /80 サブネットを使いますが、 Docker の設定でサブネットを /80 以上にも分割できます。



このセットアップでは、サブネット 2001:db8:23:42::/80 は 2001:db8:23:42:0:0:0:0 から 2001:db8:23:42:0:ffff:ffff:ffff までの範囲を eth0 に割り当て、ホスト側は 2001:db8:23:42::1 をリスニングします。サブネット 2001:db8:23:42:1::/80 は IP アドレスの範囲 2001:db8:23:42:1:0:0:0 から

2001:db8:23:42:1:ffff:ffff:ffff までを docker0 に割り当て、これがコンテナによって使われます。

NDP プロキシの使用

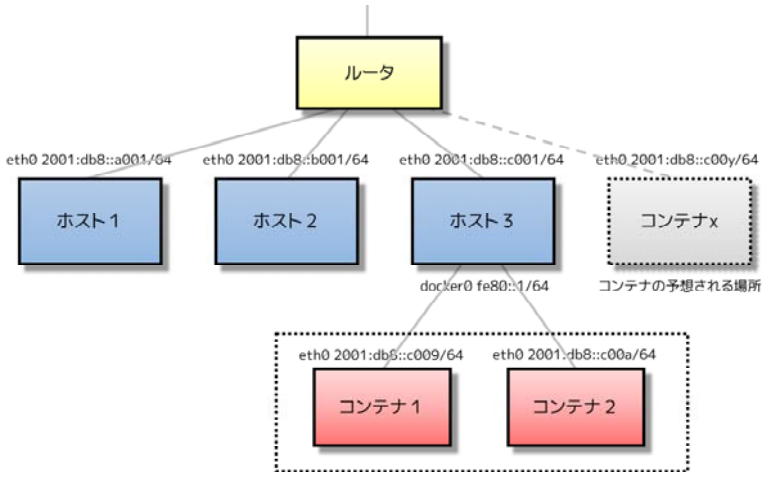
Docker ホストが IPv6 サブネットの範囲にありながら IPv6 サブネットを持たない場合、コンテナが IPv6 を経由してインターネットに接続するには、**NDP プロキシ機能（NDP proxying）**を使えます。例えば、ホストの IPv6 が 2001:db8::c001 であり、これはサブネット 2001:db8::/64 の一部です。IaaS プロバイダが 2001:db8::c000 から 2001:db8::c00f:までの IPv6 設定を許可している場合、次のように表示されます。

```
$ ip -6 addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qlen 1000
    inet6 2001:db8::c001/64 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::601:3fff:fea1:9c01/64 scope link
       valid_lft forever preferred_lft forever
```

それでは、このアドレス範囲を２つのサブネット 2001:db8::c000/125 と 2001:db8::c008/125 に分割しましょう。１つめのサブネットはホスト自身によって使われるもので、もう１つは Docker が使います。

```
docker daemon --ipv6 --fixed-cidr-v6 2001:db8::c008/125
```

Docker サブネットには、eth0 に接続するルータが管理しているサブネットが含まれているのに気を付けてください。つまり、Docker サブネットで公開される全てのデバイス（コンテナ）のアドレスは、ルータ側のサブネットから見つけることができます。つまり、ルータはこれらのコンテナと直接通信できると考えられます。



ルータは IPv6 パケットを１つめのコンテナに送ろうとしたら、すぐに**ネイバー・ソリシテーション・リクエスト（neighbor solicitation request）**を送信し、誰が 2001:db8:;c009 を持っているか訊ねます。サブネット上にアドレスが存在しなければ、誰も応答しません。コンテナはこのアドレスを Docker ホストの後ろに隠します。Docker ホストはコンテナアドレス用のネイバー・ソリシテーション・リクエストを受信したら、自分自身のデバイスがアドレスに対する責任を持っていると応答します。この処理がカーネルの **NDP Proxy** と呼ばれる機能です。有効化するには、次のコマンドを実行します。

```
$ sysctl net.ipv6.conf.eth0.proxy_ndp=1
```

これでコンテナの IPv6 アドレスを NDP プロキシ・テーブルに追加できます。

```
$ ip -6 neigh add proxy 2001:db8::c009 dev eth0
```

このコマンドはカーネルに対してネイバー・ソリシテーション・リクエストが届いているかどうか訊ねます。リクエストとは、デバイス eth- 上の IPv6 アドレス 2001:db8::c009 に対してのものです。この結果、全ての IPv6 アドレスに対するトラフィックは、Docker ホストを経由するようになります。そして、Docker ホストはコンテナのネットワークに対し、docker0 デバイスを経由し、このルーティング・テーブルに従うようにします。

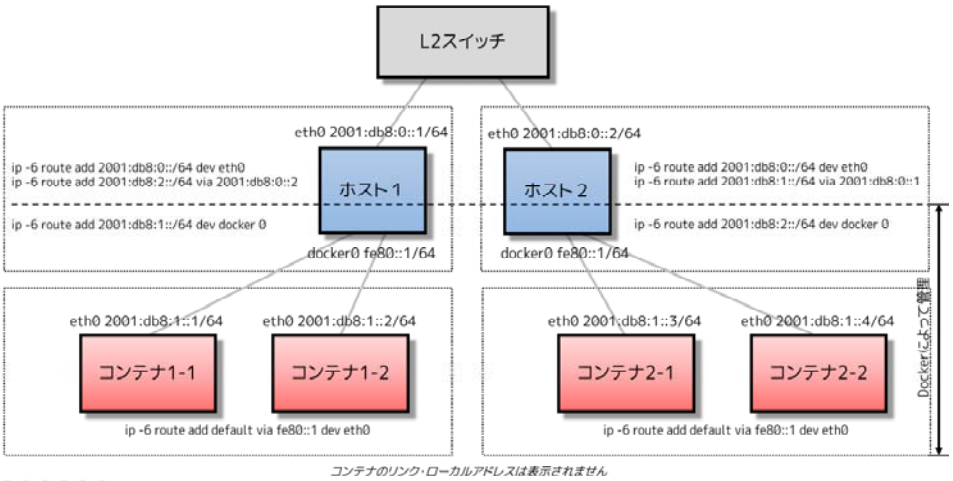
```
$ ip -6 route show
2001:db8::c008/125 dev docker0  metric 1
2001:db8::/64 dev eth0  proto kernel  metric 256
```

ip -6 neigh add proxy ... コマンドは、Docker サブネットの各 IPv6 アドレスごとに実行してきました。残念ながら、サブネットの誰がこのコマンドを実行したか把握する機能はありません。別の方法としては、ndppd のように NDP プロキシ・デーモンを使う方法があります。

Docker IPv6 クラスタ

ネットワーク環境の切り替え

到達可能な IPv6 アドレスを使い、異なるホスト上にあるコンテナ間で通信を可能にします。簡単な Docker IPv6 クラスタの例を見ていきましょう。



Docker ホストは 2001:db8:0::/64 サブネットを持ちます。ホスト 1 はコンテナに対して 2001:db8:1::/64 サブネットを自身が持つコンテナに対して提供します。そのために 3 つの経路設定をします。

- 2001:db8:0::/64 に対する全てのトラフィックは eth0 を経由する。
- 2001:db8:1::/64 に対する全てのトラフィックは docker0 を経由する。
- 2001:db8:2::/64 に対する全てのトラフィックはホスト 2 の IP アドレスを経由する。

また、ホスト 1 は OSI レイヤ 3 のルータとしても動作します。あるネットワーク・クライアントがターゲットに接続しようとする時、ホスト 1 のルーティング・テーブルを指定し、ホスト 1 がトラフィックを指定先に転送します。これはネットワーク 2001:db8::/64、2001:db8:1::/64、2001:db8:2::/64 上におけるルータとしても機能します。

ホスト 2 でも似たような設定を行います。ホスト 2 のコンテナは 2001:db8:2::/64 から IP アドレスを取得します。ホスト 2 には 3 つの経路設定があります。

- 2001:db8:0::/64 に対する全てのトラフィックは eth0 を経由する。
- 2001:db8:2::/64 に対する全てのトラフィックは docker0 を経由する。
- 2001:db8:1::/64 に対する全てのトラフィックはホスト 1 の IP アドレスを経由する。

ホスト 1 との違いは、ホスト 1 の IPv6 アドレス 2001:db8::1 には 2001:db8:1::/64 を経由するのと異なり、ホスト 2 のネットワーク 2001:db8:2::/64 は直接ホスト上の docker0 インターフェースに接続します。

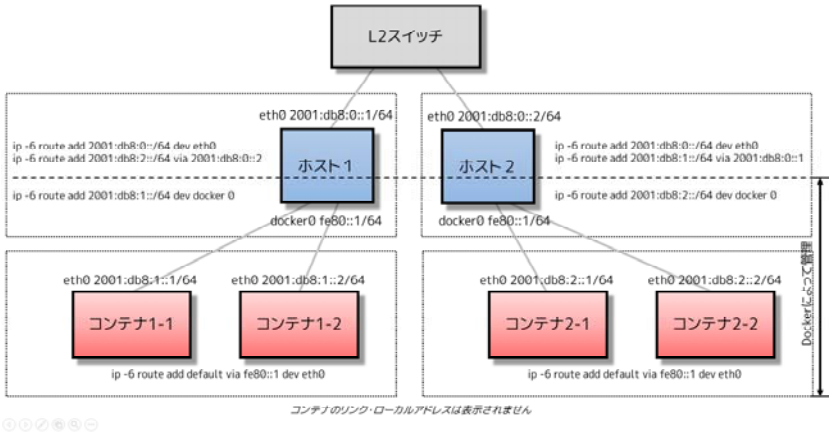
この方法は全てのコンテナが他のコンテナに対して接続できるようにします。コンテナ 1-* は同じサブネットを共有し、お互いに直接接続します。 コンテナ 1-* と コンテナ 2-* 間のトラフィックは、ホスト 1 とホスト 2 を経由します。これはこれらのコンテナが同じサブネットを共有していないためです。

ホストごとの環境の切り替え機能（switched environment）により、全てのサブネットに関する経路が判明しています。常に必要となるのは、クラスタに対するルーティング・テーブルの追加と削除のみです。

図中の各種設定のうち、点線以下は Docker が管理します。docker0 ブリッジの IP アドレス設定は、コンテナの IP アドレスを持つ Docker のサブネットに対する経路です。線から上の設定は、ユーザが個々の環境に合わせて書き換えられます。

ネットワーク経路の環境

ネットワーク環境の経路は、レイヤ 2 スイッチとレイヤ 3 ルータの関係に置き換えられます。ホストはデフォルト・ゲートウェイ（ルータ）を知っており、(Docker によって管理されている）個々のコンテナに対する経路を処理します。ルータは Docker サブネットに関する全ての経路情報も保持しています。この環境でホストの追加や削除時は、各ホストではなく、ルータ上のルーティング・テーブルを更新しなくてはいけません。



このシナリオでは、同じホスト上のコンテナは直接通信可能です。異なったホスト上にあるコンテナ間のトラフィックは、ホストとルータを経由して経路付けられます。例えば、コンテナ 1-1 から コンテナ 2-1 に対するパケットは ホスト 1 、 ルータ 、そして ホスト 2 を経由して コンテナ 2-1 に到達します。

IPv6 アドレスを短いまま維持するため、ここでは例として各ホストに /48 ネットワークを割り当てます。ホストは自身のサービスで /64 のサブネットを 1 つ使っており、もう片方は Docker です。3 つめのホストを追加する時は、2001:db8:3::/48 サブネットに対する経路をルータで行います。それから、ホスト 3 上の Docker 側で

- fixed-cidr-v6=2001:db8:3:1::/64 を設定します。

Docker コンテナのサブネットは、少なくとも /80 以上の大きさが必要なのを覚えておいてください。これは IPv6 アドレスがコンテナの MAC アドレスで終わるようにするためで、Docker レイヤにおける NDP ネイバー・キャッシュ無効化問題を防止します。もし環境に /64 があれば、 /78 はホストのサブネット用に、 /80 がコンテナ用に使われます。これにより、16 の /80 サブネットは、それぞれ 4096 のホストを使えます。

図中の各種設定のうち、点線以下は Docker が管理します。docker0 ブリッジの IP アドレス設定は、コンテナの IP

アドレスを持つ Docker のサブネットに対する経路です。線から上の設定は、ユーザが個々の環境に合わせて書き換え可能です。

5.6 その他

5.6.1 カスタム・メタデータ追加

イメージ、コンテナ、デーモンに対して**ラベル（label）**を通してメタデータを追加できます。ラベルには様々な使い方があります。例えば、メモの追加、イメージに対するライセンス情報の追加、ホストを識別するためです。

ラベルは `<キー>/<バリュー>` のペアです。Docker はラベルの値を文字列として保管します。複数のラベルを指定できますが、 `<キー>` はユニークである必要があるため、重複時は上書きされます。同じ `<キー>` を複数回指定したら、古いラベルは新しいラベルに置き換えられるため、都度値が変わります。Docker は常に指定した最新の `キー=バリュー` を使います。

デーモンのラベル機能は Docker 1.4.1 からサポートされました。コンテナとイメージに対するラベルは、Docker 1.6.0 からです。

ラベルのキーと名前空間

Docker は `キー` の使用にあたり、厳密な制約を設けていません。しかしながら、単純な `キー` であれば重複の可能性があります。例えば、`Dockerfile` の中で「architecture」ラベルを使い、CPU アーキテクチャごとにイメージを分類する場合です。

```
LABEL architecture="amd64"
LABEL architecture="ARMv7"
```

他のユーザも、同じラベルを構築時の「architecture」（構築担当者）としてラベル付けするかもしれません。

```
LABEL architecture="Art Nouveau"
```

このような名前の衝突を避けるために、Docker が推奨するのはラベルの `キー` を使うにあたり、逆ドメイン表記による名前空間を使う方法です。

- 全ての（サードパーティー製）ツールは、キーの頭に作者が管理するドメインの逆 DNS 表記を付けるべきです。例：`com.example.some-label`
- 名前空間 `com.docker.*` と `io.docker.*` と `org.dockerproject.*` は、Docker の内部利用のために予約されています。
- キーはアルファベットの小文字、ドット、ダッシュのみに統一されるべきです（例： `[a-z0-9-.]` ）。
- キーは連続したドットやダッシュを含みません。
- ネームスペース（ドット）が ない キーは CLI で使うために予約されています。これにより、エンドユーザはコマンドライン上で各々のコンテナやイメージに対してメタデータを追加する時、面倒な名前空間を指定する必要がありません。

これらは簡単なガイドラインであり、Docker は**強制しません**。しかしながら、コミュニティの利益を考えるならば、自分のラベルのキーで名前空間を使う**べき**でしょう。

構造化したデータをラベルに保存

ラベルの値は文字列で表現できるような、あらゆる種類のデータを含められます。例えば、次のような JSON ドキュメントを考えます。

```
{
    "Description": "A containerized foobar",
    "Usage": "docker run --rm example/foobar [args]",
    "License": "GPL",
    "Version": "0.0.1-beta",
    "aBoolean": true,
    "aNumber" : 0.01234,
    "aNestedArray": ["a", "b", "c"]
}
```

この構造を保管するには、文字列を 1 行に並べてラベルにできます。

```
LABEL com.example.image-specs="{\"Description\":\"A containerized foobar\",\"Usage\":\"docker run --rm
example\\/foobar
[args]\",\"License\":\"GPL\",\"Version\":\"0.0.1-beta\",\"aBoolean\":true,\"aNumber\":0.01234,\"aNes
tedArray\":[\"a\",\"b\",\"c\"]}"
```

ラベルの値に構造化データを保管できるかもしれませんが、Docker はこのデータを「普通の」文字列として扱います。これが意味するのは、Docker は深く掘り下げた（ネストする）問い合わせ（フィルタ）手法を提供しません。ツールが何らかの設定項目をフィルタする必要があれば、ツール自身で処理する機能の実装が必要です。

ラベルをイメージに追加

イメージにラベルを追加するには、Dockerfile で LABEL 命令を使います。

```
LABEL [<名前空間>.]<key>=<value> ...
```

LABEL 命令はイメージにラベルを追加し、オプションで値も追加します。 <バリュー> に空白文字列を踏む場合、ラベルをクォートで囲むかバックスラッシュを使います。

```
LABEL vendor=ACME\ Incorporated
LABEL com.example.version.is-beta=
LABEL com.example.version.is-production=""
LABEL com.example.version="0.0.1-beta"
LABEL com.example.release-date="2015-02-12"
```

また、LABEL 命令は 1 行で複数の <キー>/<バリュー> ペアの設定をサポートしています。

```
LABEL com.example.version="0.0.1-beta" com.example.release-date="2015-02-12"
```

長い行は、バックスラッシュ（\）を継続マーカーとして使い、分割できます。

```
LABEL vendor=ACME\ Incorporated \
      com.example.is-beta= \
      com.example.is-production="" \
      com.example.version="0.0.1-beta" \
      com.example.release-date="2015-02-12"
```

Docker が推奨するのは、複数のラベルを 1 つの LABEL 命令にする方法です。ラベルごとに命令するのでは、非効率なイメージになってしまいます。これは Dockerfile が LABEL 命令ごとに新しいイメージ・レイヤを作るためで

す。

ラベルの情報は docker inspect コマンドでも確認できます。

```
$ docker inspect 4fa6e0f0c678

...
"Labels": {
    "vendor": "ACME Incorporated",
    "com.example.is-beta": "",
    "com.example.is-production": "",
    "com.example.version": "0.0.1-beta",
    "com.example.release-date": "2015-02-12"
}
...

# Inspect labels on container
$ docker inspect -f "{{json .Config.Labels }}" 4fa6e0f0c678

{"Vendor":"ACME Incorporated","com.example.is-beta":"", "com.example.is-production":"",
"com.example.version":"0.0.1-beta","com.example.release-date":"2015-02-12"}

# Inspect labels on images
$ docker inspect -f "{{json .ContainerConfig.Labels }}" myimage
```

クエリ・ラベル

メタデータの保管とは別に、ラベルによってイメージとコンテナをフィルタできます。com.example.is-beta ラベルを持っている実行中のコンテナを全て表示するには、次のようにします。

```
# List all running containers that have a `com.example.is-beta` label
$ docker ps --filter "label=com.example.is-beta"
```

ラベル color が blue の全コンテナを表示します。

```
$ docker ps --filter "label=color=blue"
```

ラベル vendor が ACME の全イメージを表示します。

```
$ docker images --filter "label=vendor=ACME"
```

コンテナ・ラベル

```
docker run \
   -d \
   --label com.example.group="webservers" \
   --label com.example.environment="production" \
   busybox \
   top
```

コンテナにクエリ・ラベルをセットするには、先ほどの クエリ・ラベルのセクションをご覧ください。

デーモン・ラベル

```
docker daemon \
  --dns 8.8.8.8 \
  --dns 8.8.4.4 \
  -H unix:///var/run/docker.sock \
  --label com.example.environment="production" \
  --label com.example.storage="ssd"
```

これらのラベルは docker info によるデーモンの出力で表示されます。

```
$ docker -D info
Containers: 12
Running: 5
Paused: 2
Stopped: 5
Images: 672
Server Version: 1.9.0
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 697
 Dirperm1 Supported: true
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.19.0-22-generic
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
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:
 com.example.environment=production
 com.example.storage=ssd
```