はじめに

DockerComposeユーザガイド～基礎編

このガイドについて

Docke ドキュメント（https://docs.docker.com/）の日本語翻訳版（http://docs.docker.jp/）を元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 24 日　beta1 を公開

目次

このガイドについて 1
免責事項 1
履歴 1
1.1 概要 7
1.2 機能 8
単一ホスト上で、複数の環境を分離 8
コンテナ作成時にボリューム・データの保持 8
変更のあったコンテナのみ再作成 8
環境間で変数の共有 8
1.3 一般的な利用例 9
開発環境 9
自動テスト環境 9
単一ホストへのデプロイ 9
1.4 リリースノート 9
1.5 ヘルプを得るには 10
2.1 インストール方法 11
2.2 他のインストール方法 12
pip でインストール 12
コンテナとしてインストール 12
2.3 マスターのビルド 12
2.4 アップグレード方法 13
2.5 アンインストール方法 13
3.1 Compose を使い始める 14
事前準備 14
ステップ 1 ：セットアップ 14
ステップ 2 ：Docker イメージの作成 15
ステップ 3 ：サービスの定義 15
ステップ 4 ：Compose でアプリケーションを構築・実行 16
ステップ 5 ：他のコマンドを試す 17
3.2 Docker Compose と Django 18
プロジェクトの構成物を定義 18
Django プロジェクトの作成 19

データベースに接続 20
3.3 Docker Compose と Rasls 22
プロジェクトを定義 22
プロジェクトの構築 23
データベースに接続 23
3.4 Docker Compose と WordPress 25
プロジェクトの定義 25
プロジェクトの構築 26
ウェブ・ブラウザで WordPress を開く 26
4.1 サービスの拡張と Compose ファイル 28
複数の Compose ファイル 28
使用例 28
サービスの拡張 30
extends 設定の理解 30
使用例 31
設定の追加と上書き 32
4.2 環境ファイル 34
4.3 Swarm で Compose を使う 35
Compose の制限 35
イメージ構築 35
複数の依存関係 35
ホスト側のポートとコンテナの再作成 36
コンテナのスケジューリング 37
自動スケジューリング 37
手動スケジューリング 37
4.5 Compose をプロダクションで使う 38
Compose ファイルをプロダクション向けに書き換え 38
変更のデプロイ 38
単一サーバ上でのコンテナ実行 38
Swarm クラスタで Compose を実行する 39
4.6 Compose のネットワーク機能 40
コンテナのアップデート 40
リンク（links） 41
マルチホスト・ネットワーキング 41
カスタム・ネットワークの指定 41
デフォルト・ネットワークの設定 42
既存のネットワークを使う 42
4.7 コマンドライン補完 43

コマンドライン補完のインストール 43
Bash 43
Zsh 43
利用可能な補完 43
4.8 Compose の起動順番を制御 44
4.9 よくある質問と回答 46
サービスの起動順番を制御できますか？ 46
サービスの再作成や停止に 10 秒かかるのはどうして？ 46
同一ホスト上で Compose ファイルをコピーして、複数実行するには？ 46
up・run・start の違いは何ですか？ 46
Compose ファイルには、YAML の代わりに JSON を使えますか？ 47
データベースが起動するのを待ってからアプリケーションを起動するには？ 47
コードを入れるには COPY か ADD ですか、それともボリュームですか？ 47
Compose ファイルのサンプルはありますか？ 47
5.1 Compose ファイル・リファレンス 48
5.1.1 サービス設定リファレンス 48
build 48
context 49
dockerfile 49
args 50
cap_add, cap_drop 50
command 50
cgroup_parent 50
container_name 50
devices 51
depends_on 51
dns 51
tmpfs 52
entrypoint 52
env_file 52
environment 53
expose 53
extends 53
external_links 53
extra_hosts 54
image 54
labels 54
links 55

log_driver 56
log_opt 56
net 56
network_mode 56
networks 57
aliases 57
IPv4 アドレス、IPv6 アドレス 58
pid 58
ports 59
security_opt 59
stop_signal 59
ulimits 59
volumes, volume_driver 59
volumes_from 60
その他 61
5.1.2 ボリューム設定リファレンス 61
driver 61
driver_opts 61
external 62
5.1.3 ネットワーク設定リファレンス 62
driver 62
driver_opts 62
ipam 63
external 63
バージョン 64
アップグレード方法 65
変数の置き換え 67
5.2 docker-compose コマンド ———— 69
5.2.1 docker-compose コマンド概要 69
5.2.2 CLI 環境変数 70
COMPOSE_PROJECT_NAME 70
COMPOSE_FILE 71
COMPOSE_API_VERSION 71
DOCKER_HOST 71
DOCKER_TLS_VERIFY 71
DOCKER_CERT_PATH 71
COMPOSE_HTTP_TIMEOUT 71
5.3 Compose コマンドライン・リファレンス ———— 72

build 72
config 72
create 72
down 72
events 73
help 73
kill 73
logs 73
pause 74
port 74
ps 74
pull 74
restart 74
rm 74
run 75
scale 76
start 76
stop 76
unpause 76
up 76
5.4 リンク環境変数リファレンス ———— 78
name_PORT 78
name_PORT_num_protocol 78
name_PORT_num_protocol_ADDR 78
name_PORT_num_protocol_PORT 78
name_PORT_num_protocol_PROTO 78
name_NAME 78

1章

DockerCompose概要

1.1 概要

Docker Compose(コンポーズ) とは、複数のコンテナを使う Docker アプリケーションを、定義・実行するツールです。Compose はアプリケーションのサービスの設定に、Compose ファイルを使います。そして、コマンドを 1 つ実行するだけで、設定した全てのサービスを作成・起動します。Compose の全機能一覧について学ぶには、「機能一覧」のセクションをご覧ください。

Compose は開発環境、テスト、ステージング環境だけでなく、CI ワークフローにも適しています。それぞれの使い方の詳細を学ぶには、「一般的な利用例」のセクションをご覧ください。

Compose を使うには、基本的に 3 つのステップを踏みます。

1. アプリケーションの環境を `Dockerfile` ファイルで定義します。このファイルは、どこでも再利用可能です。
2. アプリケーションを構成する各サービスを `docker-compose.yml` ファイルで定義します。そうすることで、独立した環境を一斉に実行できるようにします。
3. 最後に、`docker-compose up` を実行したら、Compose はアプリケーション全体を起動・実行します。

`docker-compose.yml` は次のように記述します。

```
version: '2'
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
     - logvolume01:/var/log
    links:
     - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Compose に関する更に詳しい情報は、「Compose ファイル・リファレンス」のセクションをご覧ください。
Compose には、アプリケーションのライフサイクルを管理するコマンドがあります。

- サービスの開始、停止、再構築
- 実行中のサービスの状態を表示
- 実行中のサービスのストリーム・ログ出力
- サービス上で 1 回限りのコマンドを実行

1.2 機能

Compose には効率的な機能があります。

- 単一ホスト上で、複数の環境を分離
- コンテナ作成時にボリューム・データの保持
- 変更のあったコンテナのみ再作成
- 環境間で変数の共有

単一ホスト上で、複数の環境を分離

Compose は別々の環境の分離に**プロジェクト名**を使います。このプロジェクト名を次の用途で使えます。

- 開発ホスト上では、1 つの環境に対して複数のコピー作成に使います（例：プロジェクトの機能ブランチごとに、安定版のコピーを実行したい場合)。
- CI サーバ上では、お互いのビルドが干渉しないようにするため、プロジェクト名にユニークなビルド番号をセットできます。
- 共有ホストまたは開発ホスト上では、異なるプロジェクトが同じサービス名を使わないようにし、お互いを干渉しないようにします。

標準のプロジェクト名は、プロジェクトが存在するディレクトリ名です。プロジェクト名を変更するには、コマンドラインのオプションで `-p` を指定するか、環境変数の `COMPOSE_PROJECT_NAME` を指定します。

コンテナ作成時にボリューム・データの保持

Compose はサービスが使う全てのボリュームを保持（preserve）します。`docker-compose up` の実行時、以前に実行済みのコンテナが見つかれば、古いコンテナから新しいコンテナにボリュームをコピーします。この処理により、ボリューム内で作成したデータを失わないように守ります。

変更のあったコンテナのみ再作成

Compose はコンテナ作成時に使う設定情報をキャッシュします。サービスの再起動時に、内容に変更がなければ、Compose は既存のコンテナを再利用します。コンテナの再利用とは、環境をとても速く作り直せるのを意味します。

環境間で変数の共有

Compose は Compose ファイル中で、変数の使用をサポートしています。環境変数を使い、別々の環境や別々の

ユーザ向けに構成をカスタマイズできます。詳細は「環境変数」のセクションをご覧ください。

Compose ファイルは extends フィールドを使うことで、複数の Compose ファイルを作成できるように拡張できます。詳細は「extends」のセクションをご覧ください。

1.3 一般的な利用例

Compose は様々な使い方があります。一般的な利用例は、以下の通りです。

開発環境

ソフトウェアの開発時であれば、アプリケーションを別々の環境で相互にやりとりするのは重要です。Compose のコマンドライン・ツールは環境の作成と、相互のやりとりのために使えます。

Compose ファイル は、文章化と、アプリケーション全ての依存関係（データベース、キュー、キャッシュ、ウェブ・サービス、API 等）を設定するものです。Compose コマンドライン・ツールを使えば、コマンドを 1 つ（docker-compose up）実行するだけで、各依存関係に応じて 1 つまたは複数のコンテナを作成します。

同時に、開発者がプロジェクトを開始する時に役立つ機能を提供します。Compose は、複数のページにわたる「開発者向け導入手順書」を減らします。それをマシンが読み込み可能な Compose ファイルと、いくつかのコマンドで実現します。

自動テスト環境

継続的デプロイや継続的インテグレーションのプロセスにおいて重要な部分は、自動テストの実装です。自動的なエンド間（end-to-end）のテストは、テストを行う環境が必要になります。テスト実装にあたり、Compose は個々のテスト環境の作成と破棄を便利に行う手法を提供します。Compose ファイル で定義した全ての環境は、いくつかのコマンドを実行するだけで作成・破棄できます。

```
$ docker-compose up -d
$ ./run_tests
$ docker-compose stop
$ docker-compose rm -f
```

単一ホストへのデプロイ

これまでの Compose は、開発やテストにおけるワークフローに注力してきました。しかしリリースごとに、私たちはプロダクションに対応した機能を実装し続けています。Compose をリモートの Docker Engine におけるデプロイにも利用できます。Docker Engine とは、Docker Machine で自動作成された単一のマシンかもしれませんし、Docker Swarm クラスタかもしれません。

プロダクション向け機能の詳細な使い方は、「プロダクションの構成」のセクションをご覧ください。

1.4 リリースノート

Docker Compose の過去から現在に至るまでの詳細な変更一覧は、CHANGELOG[*1] をご覧ください。

*1 https://github.com/docker/compose/blob/master/CHANGELOG.md

1.5 ヘルプを得るには

Docker Compose は活発に開発中です。ヘルプが必要な場合、貢献したい場合、あるいはプロジェクトの同志と対話したい場合、私たちは多くのコミュニケーションのためのチャンネルを開いています。

バグ報告や機能リクエストは、GitHub の issue トラッカー[*1] をご利用ください。
プロジェクトのメンバーとリアルタイムに会話したければ、IRC の`#docker-compose` チャンネルにご参加ください。
コードやドキュメントの変更に貢献したい場合は、GitHub にプルリクエスト[*2] をお送りください。

より詳細な情報やリソースについては、私たちのヘルプ用ページ（英語）[*3] をご覧ください。

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

*2 https://github.com/docker/compose/pulls

*3 https://docs.docker.com/project/get-help/

2 章

Composeインストール

2.1 インストール方法

Compose は OS X、Windows、64-bit Linux で実行可能です。Compose をインストールするには、まず Docker のインストールが必要です。

Compose のインストールは、次のように実行します。

1. Docker Engine 1.7.1 以上をインストールします。

2. Docker Toolbox を使えば、Engine と Compose の両方をインストールします。そのため、Mac および Windows ユーザは、これでインストール完了です。次のステップに進んでも構いません。

3. GitHub 上にある Compose リポジトリのリリース・ページ[*1] に移動します。

4. リリース・ページの指示に従い、ターミナル上で `curl` コマンドを実行します。

もし“ Permission　denied ”エラーが表示される場合は、`/usr/local/bin` ディレクトリに対する書き込み権限がありません。その場合は Compose をスーパーユーザで実行する必要があります。`sudo -i` を実行し、２つのコマンドを実行してから `exit` します。

次の例は、コマンドの書式に関する説明です。

```
curl -L https://github.com/docker/compose/releases/download/1.6.2/docker-compose-`uname -s`-`uname -m`
> /usr/local/bin/docker-compose
```
←１行で入力

curl でのインストールに問題がある場合は、「他のインストール方法」のセクションをご覧ください。

5. バイナリに対して実行権限を追加します。

*1 https://github.com/docker/compose/releases

```
$ chmod +x /usr/local/bin/docker-compose
```

6. オプションで、bash や zsh シェルのコマンドライン補完をインストールします。

7. インストールを確認します。

```
$ docker-compose --version
docker-compose version: 1.6.2
```

2.2 他のインストール方法

pip でインストール

Compose は pypi[*1] から pip を使いインストールできます。インストールに pip を使う場合、virtualenv[*2] の利用を強く推奨します。これは多くのオペレーティング・システム上の Python システム・パッケージと、docker-compose の依存性に競合する可能性があるためです。詳しくは virtualenv チュートリアル[*3]（英語）をご覧ください。

```
$ pip install docker-compose
```

pip バージョン 6.0 以上が必要です。

コンテナとしてインストール

Compose コンテナの中でも、小さな bash スクリプトのラッパーを通すことが可能です。Compose をコンテナとして実行・インストールするには、次のようにします。

```
$ curl -L https://github.com/docker/compose/releases/download/1.6.2/run.sh \
        > /usr/local/bin/docker-compose
$ chmod +x /usr/local/bin/docker-compose
```

2.3 マスターのビルド

リリース直前（プレリリース）のビルドに興味があれば、バイナリを https://dl.bintray.com/docker-compose/master/からダウンロードできます。プレリリース版のビルドにより、リリース前に新機能を試せますが、安定性に欠けるかもしれません。

*1 https://pypi.python.org/pypi/docker-compose

*2 https://virtualenv.pypa.io/en/latest/

*3 http://docs.python-guide.org/en/latest/dev/virtualenvs/

2.4 アップグレード方法

Compose 1.2 以前からアップグレードする場合、Compose を更新後、既存のコンテナの削除・移行が必要です。これは Compose バージョン 1.3 がコンテナ追跡用に Docker ラベルを用いているためであり、ラベルを追加したものへと置き換える必要があります。

Compose は作成されたコンテナにラベルがないことを検出したら、実行を拒否し、処理停止と表示します。既存のコンテナを Compose 1.5.x 以降も使い続けたい場合（たとえば、コンテナにデータ・ボリュームがあり、使い続けたい場合）は、次のコマンドで移行できます。

```
$ docker-compose migrate-to-labels
```

あるいは、コンテナを持ち続ける必要がなければ、削除できます。Compose は新しいコンテナを作成します。

```
$ docker rm -f -v myapp_web_1 myapp_db_1 ...
```

2.5 アンインストール方法

`curl` を使って Docker Compose をインストールした場合は、次のように削除します。

```
$ rm /usr/local/bin/docker-compose
```

`pip` を使って Docker Compose をインストールした場合は、次のように削除します。

```
$ pip uninstall docker-compose
```

もし“ Permission denied ”エラーが表示される場合は、コマンドを実行する前に、docker-compose を削除するための適切な権限が必要です。強制的に削除するには sudo をあらかじめ実行してから、再度先ほどのコマンドを実行します。

3 章

Composeの使い方

3.1 Compose を使い始める

このページでは、簡単な Python ウェブ・アプリケーションを Docker Compose で実行しましょう。アプリケーションは Flask フレームワークを使い、Redis の値を増やします。サンプルでは Python を使いますが、ここでの動作概念は Python に親しくなくても理解可能です。

事前準備

既に Docker Engine と Docker Compose がインストール済みなのを確認します。Python をインストールする必要はなく、Docker イメージのものを使います。

ステップ１：セットアップ

1. プロジェクト用のディレクトリを作成します。

```
$ mkdir composetest
$ cd composetest
```

2. プロジェクト用のディレクトリに移動し、好みのエディタで app.py という名称のファイルを作成します。

```
from flask import Flask
from redis import Redis


app = Flask(__name__)
redis = Redis(host='redis', port=6379)


@app.route('/')
def hello():
    redis.incr('hits')
    return 'Hello World! I have been seen %s times.' % redis.get('hits')
```

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)
```

3. プロジェクト用のディレクトリで別の requirements.txt という名称のファイルを作成し、次の内容にします。

```
flask
redis
```

これらはアプリケーションの依存関係を定義します。

ステップ２：Docker イメージの作成

このステップでは、新しい Docker イメージを構築します。イメージには Python アプリケーションが必要とする全ての依存関係と Python 自身を含みます。

1. プロジェクト用のディレクトリの内で、Dockerfile という名称のファイルを作成し、次の内容にします。

```
FROM python:2.7
ADD . /code
WORKDIR /code
RUN pip install -r requirements.txt
CMD python app.py
```

これは Docker に対して次の情報を伝えます。

- Python 2.7 イメージを使って、イメージ構築を始める
- 現在のディレクトリ . を、イメージ内のパス /code に加える
- 作業用ディレクトリを /code に指定する
- Python の依存関係（のあるパッケージを）インストールする
- コンテナが実行するデフォルトのコマンドを python app.py にする

Dockerfile の書き方や詳細な情報については、Docker ユーザ・ガイドや Dockerfile リファレンスをご覧ください。

2. イメージを構築します。

```
$ docker build -t web .
```

このコマンドは、現在のディレクトリの内容を元にして、web という名前のイメージを構築（ビルド）します。コマンドは自動的に Dockerfile、app.py、requirements.txt を特定します。

ステップ３：サービスの定義

docker-compose.yml を使い、サービスの集まりを定義します。

1.　プロジェクト用のディレクトリに移動し、docker-compose.yml という名前のファイルを作成し、次のように追加します。

```
version: '2'
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
    depends_on:
     - redis
  redis:
    image: redis
```

この Compose 用ファイルは `web` と `redis` という２つのサービスを定義します。`web` サービスは次のように設定されます。

- 現在のディレクトリにある `Dockerfile` から構築する。
- コンテナ内の公開用（exposed）ポート 5000 を、ホストマシン上のポート 5000 に転送する。
- ホスト上のプロジェクト用のディレクトリを、コンテナ内の `/code` にマウントし、イメージを再構築しなくてもコードの変更が行えるようにする。
- web サービスを redis サービスにリンクします。

`redis` サービスには、Docker Hub レジストリから取得した最新の公開（パブリック）Redis イメージを使用します。

ステップ４：Compose でアプリケーションを構築・実行

1. プロジェクト用のディレクトリで、アプリケーションを起動します。

```
$ docker-compose up
Pulling image redis...
Building web...
Starting composetest_redis_1...
Starting composetest_web_1...
redis_1 | [8] 02 Jan 18:43:35.576 # Server started, Redis version 2.8.3
web_1   |  * Running on http://0.0.0.0:5000/
web_1   |  * Restarting with stat
```

Compose は Redis イメージを取得し、コードが動作するイメージを構築し、定義したサービスを開始します。

2. ブラウザで `http://0.0.0.0:5000/` を開き、アプリケーションの動作を確認します。

Docker を Linux で直接使っている場合は、ウェブアプリは Docker デーモンのホスト上でポート 5000 をリッスンして（開いて）います。もし `http://0.0.0.0:5000/` で接続できなければ、`http://localhost:5000` を試してください。

Mac や Windows 上で Docker Machine を使っている場合は、`docker-machine ip 仮想マシン名` を実行し、Docker ホスト上の IP アドレスを取得します。それからブラウザで `http://仮想マシンの IP:5000` を開きます。

そうすると、次のメッセージが表示されるでしょう。

```
Hello World! I have been seen 1 times.
```

3. このページを再読み込みします。

番号が増えているでしょう。

ステップ５：他のコマンドを試す

サービスをバックグラウンドで実行したい場合は、docker-compose up に -d フラグ（"デタッチド"モード用のフラグ）を付けます。どのように動作しているか見るには、docker-compose ps を使います。

```
$ docker-compose up -d
Starting composetest_redis_1...
Starting composetest_web_1...
$ docker-compose ps
Name                 Command             State       Ports
-------------------------------------------------------------------
composetest_redis_1   /usr/local/bin/run         Up
composetest_web_1     /bin/sh -c python app.py   Up      5000->5000/tcp
```

docker-compose run コマンドを使えば、サービスに対して一度だけコマンドを実行します。たとえば、web サービス上でどのような環境変数があるのかを知るには、次のようにします。

```
$ docker-compose run web env
```

docker-compose --help で利用可能な他のコマンドを確認できます。また、必要があれば bash と zsh シェル向けのコマンド補完もインストールできます。

Compose を docker-compose up -d で起動した場合は、次のようにサービスを停止して、終わらせます。

```
$ docker-compose stop
```

以上、Compose の基本動作を見てきました。

3.2 Docker Compose と Django

このクイックスタート・ガイドは Docker Compose を使い、簡単な Django/PostgreSQL アプリをセットアップします。事前に Compose のインストールが必要です。

プロジェクトの構成物を定義

このプロジェクトでは、Dockerfile と、Python 依存関係のファイル、docker-compose.yml ファイルを作成する必要があります。

1. プロジェクト用の空のディレクトリを作成します。

覚えやすい名前のディレクトリを作成します。このディレクトリがアプリケーション・イメージの内容（コンテクスト）となるものです。ディレクトリには、イメージ構築に関するリソースのみ置くべきです。

2. プロジェクト用のディレクトリに、`Dockerfile` という名称の新規ファイルを作成します。

Dockerfile はアプリケーションのイメージ内容に含まれる、1つまたは複数のイメージ構築用のコマンドを定義します。構築（ビルド）時に、コンテナの中でイメージを実行できます。Dockerfile の詳細情報については、Docker ユーザ・ガイドや Dockerfile リファレンス をご覧ください。

3. `Dockerfile` に次の内容を加えます。

```
FROM python:2.7
ENV PYTHONUNBUFFERED 1
RUN mkdir /code
WORKDIR /code
ADD requirements.txt /code/
RUN pip install -r requirements.txt
ADD . /code/
```

この `Dockerfile` は `Python 2.7` ベース・イメージを使って開始します。ベース・イメージに新しく `/code` ディレクトリ追加という変更が行われます。ベース・イメージは、`requirements.txt` ファイルで定義されている Python 依存関係のインストールという、更なる変更を定義します。

4. `Dockerfile` を保存して閉じます。

5. `requirements.txt` をプロジェクト用のディレクトリに作成します。

このファイルは `Dockerfile` の `RUN pip install -r requirements.txt` 命令で使います。

6. ファイル中に必要なソフトウェアを記述します。

```
Django
psycopg2
```

7. `requirements.txt` ファイルを保存して閉じます。

8. プロジェクト用のディレクトリに `docker-compose.yml` という名称のファイルを作成します。

`docker-compose.yml` ファイルは、アプリケーションを作るためのサービスを記述します。この例におけるサービスとはウェブサーバとデータベースです。また、Compose ファイルではサービスが利用する Docker イメージ、どのように相互にリンクするか、コンテナ内で必要となるボリュームをそれぞれ定義します。最後に `docker-compose.yml` ファイルでサービスを公開するポートを指定します。詳細な情報や動作に関しては docker-compose.yml リファレンスをご覧ください。

9. ファイルに以下の設定を追加します。

```
version: '2'
services:
  db:
    image: postgres
  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
    ports:
      - "8000:8000"
    depends_on:
      - db
```

このファイルは２つのサービスを定義しています。`db` サービスと `web` サービスです。

10. `docker-compose.yml` ファイルを保存して閉じます。

Django プロジェクトの作成

このステップでは、Django を開始するプロジェクトを作りましょう。そのためには、先の手順で構築内容を定義したイメージを作成します。

1. プロジェクト用のディレクトリに移動します。

2. Django プロジェクトを `docker-compose` コマンドを使って作成します。

```
$ docker-compose run web django-admin.py startproject composeexample .
```

これは Compose に対して、コンテナ内で `django-admin.py startproject composeexample` を実行するよう命令します。コンテナは `web` サービスのイメージと設定を使います。`web` イメージはまだ作成していませんが、`docker-compose.yml` の `build: .` 行の命令があるため、現在のディレクトリ上で構築します。

`web` サービスのイメージを構築したら、Compose はこのイメージを使い、コンテナ内で `django-admin.py startproject` を実行します。このコマンドは Django プロジェクトの代表として、Django に対してファイルとディレクトリの作成を命令します。

3. docker-compose コマンドが完了したら、プロジェクトのディレクトリ内は次のようになります。

```
$ ls -l
drwxr-xr-x 2 root   root   composeexample
-rw-rw-r-- 1 user   user   docker-compose.yml
-rw-rw-r-- 1 user   user   Dockerfile
-rwxr-xr-x 1 root   root   manage.py
-rw-rw-r-- 1 user   user   requirements.txt
```

ファイル django-admin は所有者が root として作成されました。これはコンテナが root ユーザによって実行されたからです。

Docker を Linux 上で動かしている場合は、django-admin は root の所有者として作成されます。つまり、これはコンテナが root ユーザとして実行されるのを意味します。新しいファイルの所有者を変更するには、次のように実行します。

```
sudo chown -R $USER:$USER .
```

Docker を Mac あるいは Windows 上で動かしている場合は、django-admin によって作成されたファイルも含む、全ファイルの所有者は、実行したユーザの権限です。次のように確認可能です。

```
$ ls -l
total 32
-rw-r--r--  1 user  staff  145 Feb 13 23:00 Dockerfile
drwxr-xr-x  6 user  staff  204 Feb 13 23:07 composeexample
-rw-r--r--  1 user  staff  159 Feb 13 23:02 docker-compose.yml
-rwxr-xr-x  1 user  staff  257 Feb 13 23:07 manage.py
-rw-r--r--  1 user  staff   16 Feb 13 23:01 requirements.txt
```

データベースに接続

このセクションでは、Django 用のデータベースをセットアップします。

1. プロジェクト用ディレクトリで、composeexample/settings.py ファイルを編集します。

2. DATABASES = ... を以下のものに置き換えます。

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'postgres',
        'USER': 'postgres',
        'HOST': 'db',
        'PORT': 5432,
    }
}
```

これらの設定は docker-compose.yml で指定した postgres[*1] Docker イメージによって決められているものです。

3. ファイルを保存して閉じます。

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

4. docker-compose up コマンドを実行します。

```
$ docker-compose up
Starting composepractice_db_1...
Starting composepractice_web_1...
Attaching to composepractice_db_1, composepractice_web_1
...
db_1  | PostgreSQL init process complete; ready for start up.
...
db_1  | LOG:  database system is ready to accept connections
db_1  | LOG:  autovacuum launcher started
..
web_1 | Django version 1.8.4, using settings 'composeexample.settings'
web_1 | Starting development server at http://0.0.0.0:8000/
web_1 | Quit the server with CONTROL-C.
```

これで Django アプリが Docker ホスト上のポート 8000 で動作しているでしょう。Docker Machine の仮想マシンを使っている場合は、docker-machine ip マシン名 を実行して IP アドレスを取得できます。

3.3 Docker Compose と Rasls

このクイックスタート・ガイドは Docker Compose を使い、簡単な Rails/PostgreSQL アプリをセットアップします。事前に Compose のインストールが必要です。

プロジェクトを定義

アプリケーションを構築するため、３つのファイルをセットアップしていきます。まずアプリケーションを実行する前に、Docker コンテナ内には、依存関係のある全ての準備が必要になります。そのため、コンテナ中で何が必要なのかを、正確に定義しなくてはいけません。この定義に使うのが `Dockerfile` と呼ばれるファイルです。まずはじめに、Dockerfile は次のような構成です。

```
FROM ruby:2.2.0
RUN apt-get update -qq && apt-get install -y build-essential libpq-dev nodejs
RUN mkdir /myapp
WORKDIR /myapp
ADD Gemfile /myapp/Gemfile
ADD Gemfile.lock /myapp/Gemfile.lock
RUN bundle install
ADD . /myapp
```

これはイメージ内にアプリケーションのコードを送ります。ここでは Ruby イメージを使い、Bundler や内部の依存関係を持つコンテナを作成します。Dockerfile の書き方など詳細情報については、Docker ユーザ・ガイドや Dockerfile リファレンス をご覧ください。

次に Rails を読み込むだけのブートストラップ用の `Gemfile` を作成します。`rails new` によって上書きされます。

```
source 'https://rubygems.org'
gem 'rails', '4.2.0'
```

そして `Dockerfile` の構築には、空の `Gemfile.lock` が必要です。

```
$ touch Gemfile.lock
```

最後に `docker-compose.yml` というファイルに、これら全てを結び付けます。アプリケーション構成するサービス（ここでは、ウェブサーバとデータベースです）を定義します。構成とは、使用する Docker イメージ（データベースは既製の PostgreSQL イメージを使い、ウェブ・アプリケーションは現在のディレクトリで構築します）と、必要であればどこをリンクするかや、ウェブ・アプリケーションの公開用ポートを記述します。

```
version: '2'
services:
  db:
    image: postgres
  web:
    build: .
    command: bundle exec rails s -p 3000 -b '0.0.0.0'
    volumes:
      - .:/myapp
    ports:
      - "3000:3000"
    depends_on:
      - db
```

プロジェクトの構築

これらの３つのファイルを用い、docker-compose run コマンドを使い新しい Rails スケルトン・アプリを作成します。

```
$ docker-compose run web rails new . --force --database=postgresql --skip-bundle
```

Compose はまず Dockerfile を使い web サービスのイメージを構築します。それからそのイメージを使った新しいコンテナの中で、rails new を実行します。完了すると、次のように新しいアプリが作成されています。

```
$ ls
Dockerfile   app          docker-compose.yml      tmp
Gemfile      bin          lib          vendor
Gemfile.lock config       log
README.rdoc  config.ru    public
Rakefile     db           test
```

rails new によって作成されるファイルは所有者が root でした。これはコンテナが root ユーザによって実行されたからです。新しいファイルの所有者を変更します。

```
sudo chown -R $USER:$USER .
```

新しい Gemfile から therubyracer を読み込む行をアンコメントします。これは Javascript のランタイムを入手したからです。

```
gem 'therubyracer', platforms: :ruby
```

これで新しい Gemfile ができたので、イメージを再構築する必要があります（つまり、Dockerfile の更新時、必要に応じて再起動を行うべきです)。

```
$ docker-compose build
```

データベースに接続

アプリケーションが実行可能になりましたが、まだ足りないものがあります。デフォルトでは、データベースは localhost で実行するとみなされます。そのため、db コンテナに指示しなくてはいけません。postgres イメージにデフォルトで設定されている database と username を変更する必要があります。

config/database.yml を次のように置き換えます。

```
development: &default
  adapter: postgresql
  encoding: unicode
  database: postgres
  pool: 5
  username: postgres
  password:
  host: db

test:
  <<: *default
  database: myapp_test
```

これでアプリケーションを起動できます。

```
$ docker-compose up
```

上手くいけば、次のような PostgreSQL の出力が見え、数秒後、似たような表示を繰り返します。

```
myapp_web_1 | [2014-01-17 17:16:29] INFO  WEBrick 1.3.1
myapp_web_1 | [2014-01-17 17:16:29] INFO  ruby 2.2.0 (2014-12-25) [x86_64-linux-gnu]
myapp_web_1 | [2014-01-17 17:16:29] INFO  WEBrick::HTTPServer#start: pid=1 port=3000
```

最後にデータベースを作成する必要があります。他のターミナルで、次のように実行します。

```
$ docker-compose run web rake db:create
```

以上です。これで Docker デーモン上のポート 3000 でアプリケーションが動作しているでしょう。もし Docker Machine を使っている場合は、`docker-machine ip 仮想マシン名` で Docker ホストの IP アドレスを確認できます。

3.4 Docker Compose と WordPress

Docker Compose を使えば、Docker コンテナで構築した WordPress の独立した環境を簡単に実行できます。このクイックスタート・ガイドでは、Compose のセットアップ方法と WordPress の実行方法を紹介します。事前に Compose のインストール が必要です。

プロジェクトの定義

1. プロジェクト用の空のディレクトリを作成します。

覚えやすい名前のディレクトリを作成します。このディレクトリがアプリケーション・イメージの内容（コンテクスト）となるものです。ディレクトリには、イメージ構築に関するリソースのみ置くべきです。

プロジェクト用ディレクトリには `docker-compose.yml` ファイルを置きます。このファイル自身が wordpress プロジェクトの良いスタートを切ります。

2. ディレクトリをプロジェクト用ディレクトリに変更します。

たとえば、ディレクトリ名が `my_wordpress` の場合は：

```
$ cd my-wordpress/
```

3. `docker-compose.yml` ファイルを作成します。このファイルは `wordpress` ブログと `MySQL` インスタンスを個別に起動します。`MySQL` インスタンスはデータを保持するためにボリュームをマウントします。

```
version: '2'
services:
  db:
    image: mysql:5.7
    volumes:
      - "./.data/db:/var/lib/mysql"
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    links:
      - db
    ports:
      - "8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_PASSWORD: wordpress
```

docker-compose.yml があるプロジェクトのディレクトリ内に ./.data/db　ディレクトリを自動的に作成します。wordpress がデータベースに対して更新したあらゆるデータは、このディレクトリで保持します。

プロジェクトの構築

あとは、プロジェクト用ディレクトリで docker-compose up -d を実行します。

必要なイメージを取得し、wordpress　とデータベースのコンテナを起動します。次のように画面に表示します。

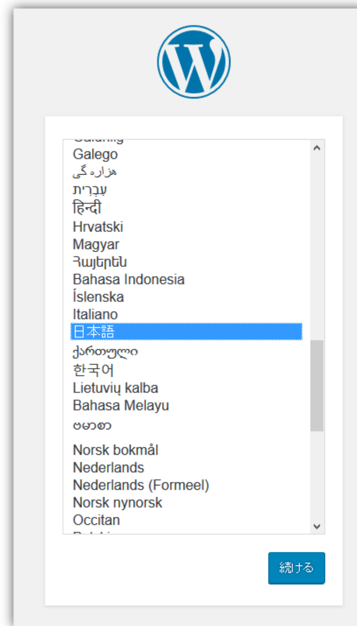
```
$ docker-compose up -d
Creating network "my_wordpress_default" with the default driver
Pulling db (mysql:5.7)...
5.7: Pulling from library/mysql
efd26ecc9548: Pull complete
a3ed95caeb02: Pull complete
...
Digest: sha256:34a0aca88e85f2efa5edff1cea77cf5d3147ad93545dbec99cfe705b03c520de
Status: Downloaded newer image for mysql:5.7
Pulling wordpress (wordpress:latest)...
latest: Pulling from library/wordpress
efd26ecc9548: Already exists
a3ed95caeb02: Pull complete
589a9d9a7c64: Pull complete
...
Digest: sha256:ed28506ae44d5def89075fd5c01456610cd6c64006addfe5210b8c675881aff6
Status: Downloaded newer image for wordpress:latest
Creating my_wordpress_db_1
Creating my_wordpress_wordpress_1
```

ウェブ・ブラウザで WordPress を開く

Docker Machine　を使っている場合は、docker-machine ip マシン名 を実行するとマシンの IP アドレスを表示します。そしてブラウザで http://マシンの IP:8000 を開きます。

この時点では WordPress は Docker ホスト上のポート 8000 で動作しています。そして、WordPress の管理者にとっては「有名な５分間のインストール」を行うだけです。

WordPress　のサイトはポート　8000　で即時利用可能になりません。なぜなら、初回読み込み時にはコンテナの初期化のために２～３分ほど必要な場合があるためです。


Galego
هزاره گی
עִבְרִית
हिन्दी
Hrvatski
Magyar
Հայերեն
Bahasa Indonesia
Íslenska
Italiano
日本語
ქართული
한국어
Lietuvių kalba
Bahasa Melayu
Norsk bokmål
Nederlands
Nederlands (Formeel)
Norsk nynorsk
Occitan
続ける

ようこそ
WordPress の有名な5分間インストールプロセスへようこそ！以下に情報を記入するだけで、世界一拡張性が高くパワフルなパーソナル・パブリッシング・プラットフォームを使い始めることができます。
必要情報
次の情報を入力してください。ご心配なく、これらの情報は後からいつでも変更できます。
サイトのタイトル
ユーザー名
ユーザー名には、半角英数字、スペース、下線、ハイフン、ピリオド、アットマーク (@) のみが使用できます。
パスワード
Jw^^WoKMUPS$Vq3gUV
隠す
強力
重要: ログイン時にこのパスワードが必要になります。安全な場所に保管してください。
メールアドレス
次に進む前にメールアドレスをもう一度確認してください。
検索エンジンでの表示
検索エンジンがサイトをインデックスしないようにする
このリクエストを尊重するかどうかは検索エンジンの設定によります。
WordPress をインストール

4章

管理

4.1 サービスの拡張と Compose ファイル

Compose は２つのファイル共有方法をサポートしています。

- Compose ファイル全体を複数の Compose ファイルを使って拡張する。
- 個々のサービスを `extends` フィールドで拡張する。

複数の Compose ファイル

デフォルトでは、Compose は２つのファイルを読み込みます。`docker-compose.yml` と、オプションの `docker-compose.override.yml`（上書き用）ファイルです。慣例として、`docker-compose.yml` には基本設定を含みます。上書きファイルとは、その名前が暗に示しているように、既存のサービスを新しいサービスに全て置き換えるものです。

もし両方のファイルに定義があれば、Compose は追加された設定の情報でルールを上書きします。

複数の上書きファイルを使いたい場合や、違った名前で上書きしたい場合は、 `-f` オプションでファイルの一覧を指定可能です。Compose はコマンドライン上で指定した順番で、ファイルを統合します。詳細は docker-compose コマンド・リファレンスの `-f` に関する情報をご覧ください。

複数の設定ファイルを使う場合、全てのパスはベースになる Compose ファイル（ `-f` で１番めに指定したファイル）からの相対パスである必要があります。この指定が必要なのは、上書き用のファイルが適切な Compose ファイル形式でなくても構わないからです。上書きファイルは設定の一部だけでも問題ありません。相対パスで指定されたサービス断片の追跡は、難しく、混乱しがちです。そのため、パスが簡単に分かるようにするため、全てのパスの定義を、ベースファイルからの相対パスにする必要があります。

使用例

このセクションでは、複数の Compose ファイルを使う２つの例をとりあげます。環境の違いにより、構成するアプリを変更する方法。それと、Compose で実行したアプリケーションに対し、管理上のタスクを実行する方法です。

異なる環境

複数のファイルを使う一般的な使用例は、開発環境のアプリ構成を、プロダクション風の環境（プロダクション

かもしれませんし、ステージングや CI かもしれません）に変更することです。環境の違いをサポートするには、Compose 設定を複数のファイルに分割します。

まず、サービスを正しく定義するベースファイルは次の通りです。

docker-compose.yml

```
web:
  image: example/my_web_app:latest
  links:
    - db
    - cache

db:
  image: postgres:latest

cache:
  image: redis:latest
```

例として、開発環境用の設定に、ホスト上の同じポートを使用し、コードをボリュームとしてマウントし、web イメージを構築するものとします。

docker-compose.override.yml

```
web:
  build: .
  volumes:
    - '.:/code'
  ports:
    - 8883:80
  environment:
    DEBUG: 'true'

db:
  command: '-d'
  ports:
    - 5432:5432

cache:
  ports:
    - 6379:6379
```

`docker-compose up` を実行したら、この上書きファイルを自動的に読み込みます。

次は、Compose を使ったアプリケーションをプロダクション環境で使えるようにします。そのために別の上書きファイルを作成します（このファイルは、異なる git リポジトリに保管されているか、あるいは異なるチームによって管理されているかもしれません)。

docker-compose.prod.yml

```
web:
  ports:
    - 80:80
  environment:
    PRODUCTION: 'true'

cache:
  environment:
    TTL: '500'
```

このプロダクション向け Compose ファイルを使ってデプロイするには、次のように実行します。

```
docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d
```

３つの全サービスがデプロイに使う設定が docker-compose.yml と docker-compose.prod.yml に含まれています（docker-compose.override.yml に含まれる開発環境はありません）。

Compose をプロダクションで使うための詳細情報は「プロダクション」のセクションをご覧ください。

管理タスク

他の一般的な使い方は、アドホックの実行や、構成アプリの１つまたは複数のサービスに対する管理タスクの実行です。ここでの例は、データベースのバックアップ実行をデモするものです。

docker-compose.yml を次のようにします。

```
web:
  image: example/my_web_app:latest
  links:
    - db

db:
  image: postgres:latest
```

docker-compose.admin.yml に、データベースをエクスポートかバックアップする新しいサービスを追加します。

```
dbadmin:
  build: database_admin/
  links:
    - db
```

通常の環境を開始するには docker-compose up -d を実行します。データベースのバックアップを行うには、docker-compose.admin.yml も使います。

```
docker-compose -f docker-compose.yml -f docker-compose.admin.yml \
    run dbadmin db-backup
```

サービスの拡張

Docker Compose の extends（拡張）キーワードは、異なったファイル間で設定を共有できるだけでなく、異なったプロジェクトでも利用可能です。拡張サービスは複数のサービスを持っている場合、一般的な設定オプションの再利用に便利です。extends を使えば、１箇所だけでなく、どこでも利用可能なサービス・オプションの共通セットを定義できます。

extends を使っても links と volumes_form はサービスを共有しません。このような例外が存在しているのは、依存性が暗黙の内に発生しないようにするためです。links と volumes_from は常にローカルで定義すべきです。そうすると、現在のファイルを読み込む時に、依存関係を明確化します。また、参照するファイルを変更したとしても、ローカルで定義する場合は壊れないようにします。

extends 設定の理解

docker-compose.yml で定義したあらゆるサービスは、次のようにして他のサービスからの拡張（extend）を宣言を宣言できます。

```
web:
  extends:
    file: common-services.yml
    service: webapp
```

これは common-services.yml ファイルで定義した webapp サービスの設定を、Compose に再利用するよう命令しています。ここでの common-services.yml は、次のようなものと仮定します。

```
webapp:
  build: .
  ports:
    - "8000:8000"
  volumes:
    - "/data"
```

この例のように、同様の docker-compose.yml の記述を行えば、web サービスに対する build、ports、volumes 設定が常に同じになります。

更に docker-compose.yml でローカル環境の設定（再設定）も行えます。

```
web:
  extends:
    file: common-services.yml
    service: webapp
  environment:
    - DEBUG=1
  cpu_shares: 5

important_web:
  extends: web
  cpu_shares: 10
```

あるいは、他のサービスから web サービスにリンクも可能です。

```
web:
  extends:
    file: common-services.yml
    service: webapp
  environment:
    - DEBUG=1
  cpu_shares: 5
  links:
    - db
db:
  image: postgres
```

使用例

個々のサービス拡張は、複数のサービスが共通の設定を持っている場合に役立ちます。以下の例では、Compose アプリはウェブ・アプリケーションとキュー・ワーカー（queue worker）の、２つのサービスを持ちます。いずれのサービスも同じコードベースを使い、多くの設定オプションを共有します。

common.yml ファイルでは、共通設定を定義します。

```
app:
  build: .
  environment:
    CONFIG_FILE_PATH: /code/config
    API_KEY: xxxyyy
  cpu_shares: 5
```

docker-compose.yml では、共通設定を用いる具体的なサービスを定義します。

```
webapp:
  extends:
    file: common.yml
    service: app
  command: /code/run_web_app
  ports:
    - 8080:8080
  links:
    - queue
    - db

queue_worker:
  extends:
    file: common.yml
    service: app
  command: /code/run_worker
  links:
    - queue
```

設定の追加と上書き

Compose は本来のサービス設定を、（訳者注：extends を使う時や、複数ファイルの読み込み時に）各所に対してコピー（引き継ぎ）します。もしも、設定オプションが元のサービスと、ローカル（直近の設定）のサービスの両方で定義された場合、ローカルの値は置き換えられるか、元の値を拡張します。

image、command、mem_limit のような単一値のオプションは、古い値が新しい値に置き換わります。

```
# 元のサービス
command: python app.py

# ローカルのサービス
command: python otherapp.py

# 結果
command: python otherapp.py
```

build と image の場合、ローカルでサービスの指定があれば、Compose は一方を破棄します。一方がオリジナルのサービスとして定義されている場合でもです。

image が build を置き換える例：

```
# 元のサービス
build: .

# ローカルのサービス
image: redis
```

```
# 結果
image: redis
```

build がイメージを置き換える例：

```
# 元のサービス
image: redis

# ローカルのサービス
build: .

# 結果
build: .
```

複数の値を持つオプション、ports、expose、external_links、dns、dns_search、tmpfs の場合、Compose は両方の値を連結します。

```
# 元のサービス
expose:
  - "3000"

# ローカルのサービス
expose:
  - "4000"
  - "5000"

# 結果
expose:
  - "3000"
  - "4000"
  - "5000"
```

environment、label、volumes、devices の場合、Compose はローカルで定義している値を優先して統合します。

```
# 元のサービス
environment:
  - FOO=original
  - BAR=original

# ローカルのサービス
environment:
  - BAR=local
  - BAZ=local

# 結果
environment:
  - FOO=original
  - BAR=local
  - BAZ=local
```

4.2 環境ファイル

Compose はファイル名 `.env` という環境ファイルを通して、デフォルトの環境変数を定義できます。このファイルは Compose ファイルと同じディレクトリに置きます。

Compose は環境ファイルの各行を **変数=値** 形式とみなします。 `#` で始まる行（例：コメント）は無視し、空白行として扱います。

実行時に環境変数を指定すると、常に `.env` ファイル中で定義した変数を上書きします。同様にコマンドラインの引数で値を指定した場合も、指定した値を優先します。

これらの環境変数は Compose ファイル内で 変数展開 のために使いますが、以下のように CLI 変数用にも使えます。

- COMPOSE_API_VERSION
- COMPOSE_FILE
- COMPOSE_HTTP_TIMEOUT
- COMPOSE_PROJECT_NAME
- DOCKER_CERT_PATH
- DOCKER_HOST
- DOCKER_TLS_VERIFY

4.3 Swarm で Compose を使う

Docker Compose と Docker Swarm は完全な統合を目指しています。つまり、Compose アプリケーションを Swarm クラスタに適用したら、単一の Docker ホスト上で展開するのと同じように動作します。
使用する Compose ファイル形式のバージョン によって、統合できる範囲が異なります。

1. バージョン 1 で `links`（リンク機能）を使う場合、アプリケーションは動作します。しかし、Swarm は全てのコンテナを 1 つのホスト上にスケジュールします。これは古いネットワーキング・システム上ではホストを横断したコンテナが扱えないためです。

2. バージョン 2 であれば、アプリケーションを変更しなくても動作するでしょう。

- 主な制限については以下をご覧ください。
- Swarm では オーバレイ・ドライバの設定や、マルチホスト・ネットワーク機能をサポートするカスタム・ドライバが利用可能です。

Docker Machine で Swarm クラスタやオーバレイ・ドライバをセットアップする方法は、「マルチホスト・ネットワーク機能を始める」セクションをご覧ください。セットアップ後にアプリケーションをデプロイするには、次のように非常にシンプルです。

```
$ eval "$(docker-machine env --swarm <name of swarm master machine>)"
$ docker-compose up
```

Compose の制限

イメージ構築

Dockerfile を使ったイメージ構築は、Swarm 上では単一ホスト上でしか行えません。そのため、構築したイメージは対象のホスト上のみに存在しており、他のノードに配布できません。
Compose を複数のノードにスケールさせる課題がある時は、自分自身で構築したレジストリ（例：Docker Hub）にイメージを push し、`docker-compose.yml` で参照させてください。

```
$ docker build -t myusername/web .
$ docker push myusername/web

$ cat docker-compose.yml
web:
  image: myusername/web

$ docker-compose up -d
$ docker-compose scale web=3
```

複数の依存関係

サービスが強制共用スケジューリング（force co-scheduling）型で複数の依存関係がある場合（以下の「自動スケジューリング」をご覧ください）、Swarm は異なったノード上でも依存関係を解決できるかもしれません。たとえば、以下の例では `foo` が必要とする `bar` と `baz` を一緒にスケジュールします。

```
version: "2"
services:
  foo:
    image: foo
    volumes_from: ["bar"]
    network_mode: "service:baz"
  bar:
    image: bar
  baz:
    image: baz
```

問題は、Swarm が最初に `bar` と `baz` が別のノードにスケジュールしてしまう可能性です（この時点ではお互いの依存性はありません)。そうならないように、`foo` を適切なノードに置く必要があります。

正常に行うためには、手動スケジューリングで、３つのサービスを同じノード上で確実に起動します。

```
version: "2"
services:
  foo:
    image: foo
    volumes_from: ["bar"]
    network_mode: "service:baz"
    environment:
      - "constraint:node==node-1"
  bar:
    image: bar
    environment:
      - "constraint:node==node-1"
  baz:
    image: baz
    environment:
      - "constraint:node==node-1"
```

ホスト側のポートとコンテナの再作成

サービスがホスト側のポートを `80:8000` のように割り当てる（マップする）場合があります。それが `docker-compose up` の初回実行時であればエラーが出るかもしれません。

```
docker: Error response from daemon: unable to find a node that satisfies
container==6ab2dfe36615ae786ef3fc35d641a260e3ea9663d6e69c5b70ce0ca6cb373c02.
```

エラーが発生する一般的なケースは、明確な割り当てのない（イメージや Compose ファイルで定義されていない）ボリュームを持つコンテナを作成する場合です。その場合はデータ領域を予約するために、Compose は Swarm に対して、前に起動したコンテナと同じノード上に新しいコンテナをスケジュールします。この結果、ポートが衝突してしまう可能性があります。

この問題に対処する２つの解決策があります。

- コンテナがボリュームをマウントできるボリューム・ドライバを使えば、ボリュームに名前を指定することで、コンテナがどのノードにスケジュールされても適切にマウントします。

Compose でサービスのボリュームに名前を付けるだけでは、Swarm に対してスケジューリングの指示を出しません。

```
version: "2"

services:
  web:
    build: .
    ports:
      - "80:8000"
    volumes:
      - web-logs:/var/log/web

volumes:
  web-logs:
    driver: custom-volume-driver
```

- 新しいコンテナを作成する前に、古いコンテナを削除したら、ボリュームの中のデータが失われます。

```
$ docker-compose stop web
$ docker-compose rm -f web
$ docker-compose up web
```

コンテナのスケジューリング

自動スケジューリング

コンテナを同じ Swarm ノード上に確実にスケジュールするための、複数のオプションがあります。オプションは次の通りです。

- network_mode: "service:..." と、network_mode: "container:..."（と、バージョン 1 のフォーマットであれば net: "container:..."）
- volumes_from
- links

手動スケジューリング

Swarm にはコンテナをどこに配置するかを制御できるようにするための、豊富なスケジューリング群と親和性の示唆（affinity hint；アフィニティ・ヒント）があります。これらはコンテナの環境を通して指定可能です。Compose では environment オプションを使って設定できます。

```
# 特定のノードにコンテナをスケジュールする
environment:
  - "constraint:node==node-1"

# 「storage」ラベルに「ssd」が設定されているノードにコンテナをスケジュールする
environment:
  - "constraint:storage==ssd"

# 「redis」イメージをダウンロード済みのコンテナにスケジュールする
environment:
  - "affinity:image==redis"
```

利用可能なフィルタと表現については、Swarm のドキュメントをご覧ください。

4.5 Compose をプロダクションで使う

開発環境で Compose を使ってアプリケーションを定義しておけば、その設定を使い、アプリケーションを CI 、ステージング、プロダクションのような異なった環境で実行できます。

アプリケーションをデプロイする最も簡単な方法は、単一サーバ上での実行です。これは開発環境で実行する方法と似ています。アプリケーションをスケールアップしたい場合には、Compose アプリを Swarm クラスタ上で実行できます。

Compose ファイルをプロダクション向けに書き換え

アプリケーションの設定を実際の環境に適用するには、ほとんどの場合で書き換えることになるでしょう。以下のような変更が必要になるかもしれません：

- コンテナのコードを外から変更できなくするため、アプリケーション・コード用に割り当てたボリュームを削除する。
- ホストに異なったポートを割り当てる。
- 異なった環境変数を割り当てる（例：冗長なログの出力を減らす、あるいは、メールの送信を有効化）
- 再起動ポリシーを指定し（例：`restart: always`）、停止時間を減らす
- 外部サービスの追加（例：ログ収集）

このような理由のため、`production.yml` のような追加 Compose ファイルを使い、プロダクションに相応しい設定を定義したくなるでしょう。この設定ファイルには、元になった Compose ファイルからの変更点のみ記述できます。追加の Compose ファイルは、元の `docker-compose.yml` の設定を上書きする新しい設定を指定できます。

２つめの設定ファイルを使うには、Compose で `-f` オプションを使います。

```
$ docker-compose -f docker-compose.yml -f production.yml up -d
```

詳細は例は 複数の Compose ファイルを使用のセクションをご覧ください。

変更のデプロイ

アプリケーションのコードを変更した時は、イメージを再構築し、アプリケーションのコンテナを作り直す必要があります。`web` という名称のサービスを再デプロイするには、次のように実行します。

```
$ docker-compose build web
$ docker-compose up --no-deps -d web
```

これは、まず `web` イメージを再構築するために（コンテナを）停止・破棄します。それから `web` サービスのみ再作成します。`--no-deps` フラグを使うことで、Compose が web に依存するサービスを再作成しないようにします。

単一サーバ上でのコンテナ実行

Compose を使い、アプリケーションをリモートの Docker ホスト上にデプロイできます。この時、適切な環境変数 `DOCKER_HOST`、`DOCKER_TLS_VERIFY`、`DOCKER_CERT_PATH` を使います。このような処理は、Docker Machine を使うことで、ローカルやリモートの Docker ホストの管理を非常に簡単にします。リモートにデプロイする必要がなくても、お勧めです。

環境変数を設定するだけで、追加設定なしに `docker-compose` コマンドが普通に使えます。

Swarm クラスタで Compose を実行する

Docker　Swarm とは、Docker 独自のクラスタリング・システムで、単一の Docker ホスト向けと同じ API を持っています。つまり、Compose を Swarm インスタンスも同様に扱えますので、アプリケーションを複数のホスト上で実行できることを意味します。

4.6 Compose のネットワーク機能

このドキュメントが適用されるのは Compose ファイル・フォーマットのバージョン２ を使う場合です。ネットワーク機能はバージョン１（過去）の Compose ファイルではサポートされていません。

デフォルトでは、Compose はアプリケーションに対して ネットワーク を１つ設定します。各コンテナ上のサービスはデフォルト・ネットワークに参加したら、同一ネットワーク上の他のコンテナから接続できるようになります。また、ホスト名とコンテナ名でも発見可能になります。

アプリケーション用のネットワークには、”プロジェクト名” と同じ名前が割り当てられます。プロジェクト名とは、作業している基準の（ベースとなる）ディレクトリ名です。このプロジェクト名は `--project-name` フラグか `COMPOSE_PROJECT_NAME` 環境変数で変更できます。

例として、アプリケーションを置いたディレクトリ名を `myapp` とし、`docker-compose.yml` は次のような内容とします。

```
version: '2'

services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
```

docker-compose up を実行したら、次のように動作します。

1. `myapp_default` という名称のネットワークを作成します。
2. `web` 設定を使ったコンテナを作成します。これをネットワーク `myapp_default` に対して、`web` という名称で追加します。
3. `db` 設定を使ったコンテナを作成します。これをネットワーク `myapp_default` に対して、`db` という名称で追加します。

各コンテナは、これでホスト名を `web` あるいは `db` で名前解決することにより、コンテナに割り当てられた IP アドレスが分かります。たとえば、`web` アプリケーションのコードが URL `postgres://myapp_db_1:5432` にアクセスできるようになり、PostgreSQL データベースを利用開始します。

`web` はポートの割り当てを明示しているため、Docker ホスト側のネットワーク・インターフェース上からも、ポート 8000 を通して外からアクセス可能です。

コンテナのアップデート

サービスの設定を変更するには、`docker-compose up` を実行して、古いコンテナの削除と新しいコンテナをネットワーク下で起動します。IP アドレスは異なりますが、ホスト名は同じです。実行中のコンテナはその名前で名前解決が可能になり、新しい IP アドレスで接続できますが、古い IP アドレスは機能しなくなります。

もし古いコンテナに対して接続しているコンテナがあれば、切断されます。この状況検知はコンテナ側の責任であり、名前解決を再度行い再接続します。

リンク（links）

Docker のリンク（link）は、一方通行の単一ホスト上における通信システムです。この機能は廃止される可能性があり、アプリケーションはネットワーク機能を使うようにアップデートすべきです。多くの場合は、docker-compose.yml で link セクションを削除するだけです。

リンク機能（links）とは、他のサービスから到達可能なエイリアス（別名）を定義するものです。サービス間で通信するために必要ではありません。すなわち、デフォルトでは、あらゆるサービスはサービス名を通して到達できます。以下の例では、web から db に到達するには、ホスト名の db と（エイリアスの）database が使えます。

```
version: '2'
  services:
    web:
      build: .
      links:
        - "db:database"
    db:
      image: postgres
```

詳しい情報は links リファレンスをご覧ください。

マルチホスト・ネットワーキング

Compose アプリケーションを Swarm クラスタにデプロイする時に、ビルトインの overlay ドライバを使い、複数のホストを通してコンテナ間の通信を可能にできます。そのためにアプリケーションのコードや Compose ファイルを書き換える必要はありません。

Swarm クラスタのセットアップの仕方は、複数のホストでネットワーク機能を使う方法 を参考にしてください。デフォルトは overlay ドライバを使いますが、任意のドライバを指定可能です。詳しくは後述します。

カスタム・ネットワークの指定

デフォルトのアプリケーション用のネットワークを使う代わりに、自分で任意のネットワーク指定が可能です。そのためには、トップレベルの networks キーを（Compose ファイルで）使います。これにより、より複雑なトポロジのネットワーク作成や、 カスタム・ネットワーク・ドライバ やオプションを指定できます。また、Compose によって管理されない、外部に作成したネットワークにサービスも接続できます。

サービス・レベルの networks キーを使うことで、各サービスがどのネットワークに接続するか定義できます。このキーは トップ・レベル の networks キー直下にあるエントリ一覧から名前を参照するものです。

以下の Compose ファイルの例では、２つのカスタム・ネットワークを定義しています。proxy サービスと db サービスは独立しています。これは共通のネットワークに接続していないためです。app のみが両方と通信できます。

```
version: '2'

services:
  proxy:
    build: ./proxy
    networks:
      - front
  app:
    build: ./app
    networks:
      - front
      - back
  db:
```

```
      image: postgres
      networks:
        - back

networks:
  front:
    # Use a custom driver
    driver: custom-driver-1
  back:
    # Use a custom driver which takes special options
    driver: custom-driver-2
    driver_opts:
      foo: "1"
      bar: "2"
```

ネットワークでは、接続したネットワーク上で IPv4 アドレスと IPv6 アドレスの両方、またはいずれかを設定できます。

ネットワーク設定オプションに関する詳しい情報は、以下のリファレンスをご覧ください。

- トップ・レベル networks キー
- サービス・レベル networks キー

デフォルト・ネットワークの設定

自分でネットワークを定義する場合、しない場合どちらでも、アプリケーション全体に適用できるデフォルトのネットワークを networks の直下の default エントリで定義できます。

```
version: '2'

services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres

networks:
  default:
    # Use a custom driver
    driver: custom-driver-1
```

既存のネットワークを使う

コンテナを既存のネットワークに接続したい場合は、external オプションを使います。

```
networks:
  default:
    external:
      name: my-pre-existing-network
```

[プロジェクト名]_default という名称のネットワークを作成しようとしなくても、Compose は my-pre-existing-network という名称のネットワークを探し出し、コンテナのアプリケーションを接続できます。

4.7 コマンドライン補完

Compose は bash と zsh シェル向けのコマンド補完機能を搭載しています。

コマンドライン補完のインストール

Bash

bash 補完（completion）がインストールされているかどうか確認します。現在の Linux が最小インストールでなければ、bash 補完は利用可能です。Mac は `brew install bash-completion` でインストールします。

次のようにして、補完スクリプトを `/etc/bash_completion.d/`（Mac は `/usr/local/etc/bash_completion.d/`）に置きます。

```
curl -L https://raw.githubusercontent.com/docker/compose/$(docker-compose version
--short)/contrib/completion/bash/docker-compose > /etc/bash_completion.d/docker-compose  ←1行で入力
```

次回ログイン時から補完機能が利用可能になります。

Zsh

補完スクリプトを `/path/to/zsh/completion` や、`~/.zsh/completion/` に置きます。

```
mkdir -p ~/.zsh/completion
curl -L https://raw.githubusercontent.com/docker/compose/$(docker-compose version
--short)/contrib/completion/zsh/_docker-compose > ~/.zsh/completion/_docker-compose  ←1行で入力
```

`$fpath` には `~/.zshrc` ディレクトリを追加します。

```
fpath=(~/.zsh/completion $fpath)
```

`compinit` で読み込まれて `~/.zshrc` に追加されているか確認します。

```
autoload -Uz compinit && compinit -i
```

それからシェルを再読み込みします。

```
exec $SHELL -l
```

利用可能な補完

コマンドラインの補完は、入力する内容に依存します。

- 利用可能な docker-compose コマンド
- 個々のコマンドで利用可能なオプション
- 指定した状態にあるサービス名（例：サービスが実行中、停止中、サービスの元になったイメージ、あるいは Dockerfile の元となるサービス）。`docker-compose scale` の補完では、サービス名に自動的に "=" を追加します。
- 選択したオプションに対する引数。たとえば `docker-compose kill -s` は `SIGHUP` や `SIGUSR1` のようなシグナルを補完します。

Compose をより速く・入力ミス（typo）なく楽しんで使いましょう！

4.8 Compose の起動順番を制御

depends_on オプションを使えば、サービスの起動順番を制御できます。Compose は常に依存関係に従ってコンテナを起動しようとします。依存関係とは、depends_on、links、volumes_form、network_mode: "サービス:..." を指定している場合です。

しかしながら、Compose はコンテナの準備が「整う」まで待ちません（つまり、特定のアプリケーションが利用可能になるまで待ちません)。単に起動するだけです。これには理由があります。

たとえば、データベースの準備が整うまで待つのであれば、そのことが分散システム全体に対する大きな問題になり得ます。プロダクションでは、データベースは利用不可能になったり、あるいは別のホストに移動したりする場合があるでしょう。アプリケーションは、障害発生に対して復旧する必要があるためです。

データベースに対する接続が失敗したら、アプリケーションは再接続を試みるように扱わなくてはいけません。アプリケーションは再接続を試みるため、データベースへの接続を定期的に行う必要があるでしょう。

この問題を解決するベストな方法は、アプリケーションのコード上で解決することです。起動時に、あるいは何らかの理由によって接続できない場合にです。しかしながら、このレベルの復旧が必要なければ、ラッパー用のスクリプトを書くことでも対処できます。

- wait-for-it[*1] や dockerize[*2] のようなツールを使います。これらはラッパー用のスクリプトであり、アプリケーションのイメージに含めることができます。また特定のホスト側のポートに対して、TCP 接続を受け入れ可能です。

アプリケーションのイメージに適用するためには、Dockerfile の CMD 命令でラップできるように docker-compose.yml の entrypoint を設定します。

```
version: "2"
services:
  web:
    build: .
    ports:
      - "80:8000"
    depends_on:
      - "db"
    entrypoint: ./wait-for-it.sh db:5432
  db:
    image: postgres
```

- アプリケーションが独自にヘルスチェックを行えるよう、スクリプトをラッパーすることも可能です。たとえば、Postgres コマンドが使えるようになるまで待ちたい場合を考えてみましょう。

```
#!/bin/bash

set -e

host="$1"
shift
cmd="$@"
```

*1 https://github.com/vishnubob/wait-for-it

*2 https://github.com/jwilder/dockerize

```
until psql -h "$host" -U "postgres" -c '\l'; do
  >&2 echo "Postgres is unavailable - sleeping"
  sleep 1
done

>&2 echo "Postgres is up - executing command"
exec $cmd
```

このラッパー・スクリプトの例を使うには、`entrypoint: ./wait-for-postgres.sh db` と指定します。

4.9 よくある質問と回答

あなたの質問がここになければ、freenode IRC の #docker-compose にあるコミュニティに、質問を気軽に投げてください。

サービスの起動順番を制御できますか？

はい、詳細は「Compose の起動順番を制御」のセクションをご覧ください。

サービスの再作成や停止に 10 秒かかるのはどうして？

Compose の停止（stop）とはコンテナに SIGTERM を送信して停止することです。 デフォルトのタイムアウトは 10 秒間です。タイムアウトしたら、コンテナを強制停止するために SIGKILL を送信します。タイムアウトで待っているとは、つまり、コンテナが SIGTERM シグナルを受信しても停止しないのを意味します。

これは「コンテナがプロセスのシグナルをどう扱うか」[*1]（英語）の問題で言及されています。

この問題を解決するには、以下のことを試してください。

- Dockerfile の CMD と ENTRYPOINT 命令で JSON 形式を使用する。

たとえば、["program", "arg1", "arg2"] の形式を使うのであり、 "program arg1 arg2" の形式を使いません。後者の文字列で指定すると、Docker はプロセスの実行に bash を使います。すると Docker は適切にシグナルを扱えません。Compose では常に JSON 形式を使っていれば、Compose ファイル上のコマンドやエントリを書き換える心配は不要です。

- 可能であれば、アプリケーションが SIGTERM シグナルを確実に扱えるように書き換えます。

- アプリケーションを書き換え不可能な場合は、アプリケーションを（s6[*2] のような）軽量な init システムを使うか、あるいはシグナルを（dumb-init[*3] や tini[*4] などで）プロキシします。いずれかのラッパーを使い、SIGTERM シグナルを適切に扱います。

同一ホスト上で Compose ファイルをコピーして、複数実行するには？

Compose での作成時は、プロジェクト名をユニークな識別子として使います。これはプロジェクト内の全てのコンテナや他のリソースに使います。プロジェクトをコピーして複数実行するには、 -p コマンドライン・オプションを使って任意のプロジェクト名を指定するか、あるいは COMPOSE_PROJECT_NAME 環境変数を使います。

up・run・start の違いは何ですか？

一般的には docker-compose up が使われるでしょう。up を使うと docker-compose.yml ファイル中で定義したサービスの開始または再起動を行います。デフォルトは「アタッチド」モードであり、全てのコンテナのログが画面上に表示されます。「デタッチド」モード（ -d ）では、Compose はコンテナを実行すると終了しますが、コンテ

---

*1 https://medium.com/@gchudnov/trapping-signals-in-docker-containers-7a57fdda7d86

*2 http://skarnet.org/software/s6/

*3 https://github.com/Yelp/dumb-init

*4 https://github.com/krallin/tini

ナは後ろで動き続けます。

docker-compose run コマンドは「ワンオフ」(one-off；１つだけ、偶発的）　または「アドホック」(adhoc；臨時）なタスクの実行に使います。実行するにはサービス名の指定が必要であり、特定のサービス用のコンテナを起動し、かつ依存関係のあるコンテナも起動します。run の利用時は、テストの実行であったり、データ・ボリューム・コンテナに対するデータの追加・削除といった管理タスクです。run コマンドは実際には docker run -ti を処理しており、コンテナに対してインタラクティブなターミナルを開き、コンテナのプロセスが終了すると、その時点の該当する終了コードを返します。

docker-compose start コマンドは既に作成済みのコンテナの再起動には便利です。しかし止まっているコンテナを起動するだけであり、新しいコンテナは作成しません。

Compose ファイルには、YAML の代わりに JSON を使えますか？

はい、YAML は JSON のスーパーセットです 。そのため、あらゆる JSON ファイルは有効な YAML でもあります。Compose ファイルに JSON を使いたい場合は、ファイル名で（ .json を）指定します。実行例：

```
docker-compose -f docker-compose.json up
```

データベースが起動するのを待ってからアプリケーションを起動するには？

残念ながら、正統な理由がなければ Compose はそのように処理できません。

データベースが準備するまで待たせることにより、分散システムにおいて非常に大きな問題となる可能性があります。プロダクション環境においては、データベースが使えなくなるか、あるいは他のホストに移動する場合があるでしょう。アプリケーションは、この種の障害に対する回復力を必要とします。

そのためには、アプリケーションがデータベースとの通信ができなくなっても、再度接続を試みるでしょう。もしアプリケーションのリトライが失敗したら、データベースに対する接続性は失われたと考えるべきです。

アプリケーションが正常になるまで待つためには、ヘルスチェックを実装する必要があります。ヘルスチェックはアプリケーションに対してリクエストを送り、ステータス・コードの応答が正常化どうかを確認します。もし成功しなければ、短時間待った後、再試行します。何度もタイムアウトをするようであれば、チェックを停止し、失敗を報告します。

もしアプリケーションに対する実行テストが必要であれば、ヘルスチェックを実行できます。ヘルスチェックの応答が正常であれば、テストを実行可能になります。

コードを入れるには COPY か ADD ですか、それともボリュームですか？

コードをイメージにコピーするには、Dockerfile の COPY または ADD 命令が使えます。これは Docker イメージのコードを置き換える場合に便利です。たとえば、コードを別の環境（プロダクション、CI 、等）に送りたい場合です。

コードを変更したい場合、すぐに反映したい場合は volume を使うべきでしょう。たとえば、コードをデプロイする場面で、サーバがホット・コード・リロードやライブ・リロードをサポートしている場合です。

両方の命令を使いたい場合があるかもしれません。開発環境上において、イメージに対してコードを追加する場合は COPY を使い、Compose ファイルにコードを含める場合は volume を使えます。ボリュームを使えばイメージの中にあるディレクトリの情報を上書きします。

Compose ファイルのサンプルはありますか？

GitHub 上に Compose ファイルのサンプルがたくさんあります。

https://github.com/search?q=in%3Apath+docker-compose.yml+extension%3Ayml&type=Code

5 章

Composeリファレンス

5.1 Compose ファイル・リファレンス

Compose ファイルは YAML ファイルであり、サービス（services）、ネットワーク（networks）、ボリューム（volumes）を定義します。Compose ファイルのデフォルトのパスは `./docker-compose.yml` です。

サービスの定義では、各コンテナをサービスとして定義できます。このサービスを起動する時、コマンドラインの `docker run` のパラメータのような指定が可能です。同様に、ネットワークやボリュームの定義も `docker network create` や `docker volume create` と似ています。

`docker run` では、Dockerfile で指定したオプション（例：`CMD`、`EXPOSE`、`VOLUME`、`ENV`）はデフォルトとして尊重されます。そのため、 `docker-compose.yml` で再び指定する必要はありません。

Bash の `${変数}` の構文のように、環境変数を使って設定を行えます。詳しくは「変数の置き換え」のセクションをご覧ください。

5.1.1 サービス設定リファレンス

Compose ファイルの形式には、バージョン１（過去のフォーマットであり、ボリュームやネットワークをサポートしていません）とバージョン２（最新版）という２つのバージョンが存在します。詳しい情報はバージョンに関するドキュメントをご覧ください。

（Docker Compose の）**サービス**定義用にサポートされている設定オプションの一覧を、このセクションで扱います。

build

構築時に適用するオプションを指定します。

`build` で指定できるのは、構築用コンテクストのパスを含む文字列だけでなく、`context` の配下にある特定のもの（オブジェクト）や、`dockerfile` のオプションと**引数**を指定できます。

```
build: ./dir

build:
  context: ./dir
  dockerfile: Dockerfile-alternate
```

```
  args:
    buildno: 1
```

build だけでなく image も指定できます。Compose は image で指定したタグを使い、構築したイメージをタグ付けします。

```
build: ./dir
image: webapp
```

これは ./dir で構築したイメージを webapp としてタグ付けしています。

バージョン１のフォーマットでは、**build** の使い方が異なります：

- build: . の文字列のみ許可されています。オブジェクトは指定できません。
- build と image は同時に使えません。指定するとエラーになります。

context

context はバージョン２のフォーマットのみで利用可能です。バージョン１では build をお使いください。

コンテクスト（訳者注：内容物の意味）には Dockerfile があるディレクトリのパスや Git リポジトリの URL を指定します。
値に相対パスを指定したら、Compose ファイルのある場所を基準とした相対パスとして解釈します。また、指定したディレクトリが構築コンテクストとなり、Docker デーモンに送信します。
Compose は生成時の名前で構築・タグ付けし、それがイメージとなります。

```
build:
  context: ./dir
```

dockerfile

Dockerfile の代わりになるものです。
Compose は構築時に別のファイルを使えます。構築時のパスも指定する必要があります。

```
build:
  context: .
  dockerfile: Dockerfile-alternate
```

バージョン１のフォーマットでは、**dockerfile** の使い方が異なります：

- build と dockerfile は並列であり、サブオプションではありません。

  build: . dockerfile: Dockerfile-alternate
- dockerfile と image を同時に使えません。使おうとしてもエラーになります。

args

対応しているのはバージョン２のファイル形式のみです。

構築時に build のオプション（args）を追加します。配列でも辞書形式（訳者注：「foo=bar」の形式）も指定できます。ブール演算子（true、false、yes、no）を使う場合はクォートで囲む必要があります。そうしませんと YAML パーサは True か False か判別できません。

構築時に引数のキーとして解釈する環境変数の値は、Compose を実行するマシン上のみです。

```
build:
  args:
    buildno: 1
    user: someuser

build:
  args:
    - buildno=1
    - user=someuser
```

cap_add, cap_drop

コンテナのケーパビリティ（capabilities）を追加・削除します。ケーパビリティの一覧は `man 7 capabilities` をご覧ください。

```
cap_add:
  - ALL

cap_drop:
  - NET_ADMIN
  - SYS_ADMIN
```

command

デフォルトのコマンドを上書きします。

```
command: bundle exec thin -p 3000
```

これは Dockerfile の書き方に似せることもできます。

```
command: [bundle, exec, thin, -p, 3000]
```

cgroup_parent

コンテナに対し、オプションの親グループを指定します。

```
cgroup_parent: m-executor-abcd
```

container_name

デフォルトで生成される名前の代わりに、カスタム・コンテナ名を指定します。

```
container_name: my-web-container
```

Docker コンテナ名はユニークである必要があります。そのため、カスタム名を指定時、サービスは複数のコンテナにスケールできなくなります。

devices

デバイス・マッピングの一覧を表示します。docker クライアントで作成する際の `--device` 同じ形式を使います。

```
devices:
  - "/dev/ttyUSB0:/dev/ttyUSB0"
```

depends_on

サービス間の依存関係を指定したら、２つの効果があります。

- `docker-compose up` を実行したら、依存関係のある順番に従ってサービスを起動します。以下の例では、`web` を開始する前に `db` と `rails` を実行します。
- `docker-compose up` サービス（の名称）を実行したら、自動的にサービスの依存関係を処理します。以下の例では、`docker-compose up web` を実行したら、`db` と `redis` も作成・起動します。

簡単なサンプル：

```
version: '2'
services:
  web:
    build: .
    depends_on:
      - db
      - redis
  redis:
    image: redis
  db:
    image: postgres
```

depends_on では、web の実行にあたり、db と radis の準備が整うのを待てません。待てるのはコンテナを開始するまでです。サービスの準備が整うまで待たせる必要がある場合は、「起動順番の制御」関するドキュメントで、問題への対処法や方針をご確認ください。

dns

DNS サーバの設定を変更します。単一の値、もしくはリストになります。

```
dns: 8.8.8.8
dns:
  - 8.8.8.8
  - 9.9.9.9

dns_search
```

DNS の検索ドメインを変更します。単一の値、もしくはリストになります。

```
dns_search: example.com
dns_search:
  - dc1.example.com
  - dc2.example.com
```

tmpfs

コンテナ内にテンポラリ・ファイルシステムをマウントします。単一の値もしくはリストです。

```
tmpfs: /run
tmpfs:
  - /run
  - /tmp
```

entrypoint

デフォルトの entrypoint を上書きします。

```
entrypoint: /code/entrypoint.sh
```

entrypoint は Dockerfile のように列挙できます。

```
entrypoint:
    - php
    - -d
    - zend_extension=/usr/local/lib/php/extensions/no-debug-non-zts-20100525/xdebug.so
    - -d
    - memory_limit=-1
    - vendor/bin/phpunit
```

env_file

ファイル上の定義から環境変数を追加します。単一の値、もしくはリストになります。

Compose ファイルを `docker-compose -f ファイル名` で指定する場合は、`env_file` ファイルは指定したディレクトリに対する相対パスとみなします。

環境変数で指定されている値は、`environment` で上書きできます。

```
env_file: .env

env_file:
  - ./common.env
  - ./apps/web.env
  - /opt/secrets.env
```

Compose は各行を `変数=値` の形式とみなします。 `#` で始まる行（例：コメント）は無視され、空白行として扱います。

```
# Rails/Rack 環境変数を設定
RACK_ENV=development
```

environment

環境変数を追加します。配列もしくは辞書形式（dictionary）で指定できます。boolean 値は true、false、yes、no のいずれかであり、YML パーサによって True か False に変換されるよう、クォート（‘ 記号）で囲む必要があります。

キーだけの環境変数は、Compose の実行時にマシン上で指定するものであり、シークレット（訳注：AP 鍵などの秘密情報）やホスト固有の値を指定するのに便利です。

```
environment:
  RACK_ENV: development
  SHOW: 'true'
  SESSION_SECRET:

environment:
  - RACK_ENV=development
  - SHOW=true
  - SESSION_SECRET
```

expose

ホストマシン上で公開するポートを指定せずに、コンテナの公開（露出）用のポート番号を指定します。これらはリンクされたサービス間でのみアクセス可能になります。内部で使うポートのみ指定できます。

```
expose:
 - "3000"
 - "8000"
```

extends

現在のファイルから別のファイルにサービスを拡張するもので、設定のオプションを追加します。

他の設定用のキーと一緒にサービスを `extends`（拡張）できます。`extends` 値には `service` の定義が必要であり、オプションで `file` キーを指定します。

```
extends:
  file: common.yml
  service: webapp
```

サービスを拡張する `service` の名前とは、たとえば `web` や `database` です。`file` はサービスを定義する Compose 設定ファイルの場所です。

`file` を省略したら、Compose は現在の設定ファイル上からサービスの定義を探します。`file` の値は相対パスまたは絶対パスです。相対パスを指定したら、Compose はその場所を、現在のファイルからの相対パスとして扱います。

自分自身を他に対して拡張するサービス定義ができます。拡張は無限に可能です。Compose は循環参照をサポートしておらず、もし循環参照があれば `docker-compose` はエラーを返します。

extends に関するより詳細は、extends のドキュメントをご覧ください。

external_links

対象の `docker-compose.yml` の外にあるコンテナだけでなく、Compose の外にあるコンテナとリンクします。特に、コンテナが共有サービスもしくは一般的なサービスを提供している場合に有用です。`external_links` でコンテナ名とエイリアスを指定すると（ コンテナ名:エイリアス名 ）、link のように動作します。

```
external_links:
 - redis_1
 - project_db_1:mysql
 - project_db_1:postgresql
```

バージョン２のファイル形式 を使う時、外部に作成したコンテナと接続する必要があれば、接続先のサービスは対象ネットワーク上に少なくとも１つリンクする必要があります。

extra_hosts

ホスト名を割り当てます。これは docker クライアントで `--add-host` パラメータを使うのと同じものです。

```
extra_hosts:
 - "somehost:162.242.195.82"
 - "otherhost:50.31.209.229"
```

コンテナ内の /etc/hosts に IP アドレスとホスト名のエントリが追加されます。例：

```
162.242.195.82  somehost
50.31.209.229   otherhost
```

image

コンテナを実行時に元となるイメージを指定します。リポジトリ名・タグあるいはイメージ ID の一部を指定できます。

```
image: redis
image: ubuntu:14.04
image: tutum/influxdb
image: example-registry.com:4000/postgresql
image: a4bc65fd
```

イメージが存在していなければ、Compose は pull（取得）を試みます。しかし build を指定している場合は除きます。その場合、指定されたタグやオプションを使って構築します。

バージョン１のファイル形式では、`build` と `image` を同時に使えません。実行しようとしてもエラーが出ます。

labels

Docker ラベルを使いコンテナにメタデータを追加します。配列もしくは辞書形式で追加できます。
他のソフトウェアとラベルが競合しないようにするため、DNS 逆引き記法の利用を推奨します。

```
labels:
  com.example.description: "Accounting webapp"
  com.example.department: "Finance"
  com.example.label-with-empty-value: ""

labels:
  - "com.example.description=Accounting webapp"
  - "com.example.department=Finance"
```

```
    - "com.example.label-with-empty-value"
```

links

コンテナを他のサービスとリンクします。サービス名とリンク用エイリアスの両方を指定できます（ サービス名:エイリアス名 ）。あるいはサービス名だけの指定もできます（このサービス名はエイリアス名としても使われます)。

```
links:
 - db
 - db:database
 - redis
```

リンクするサービスのコンテナは、エイリアスとして認識できるホスト名で到達（接続）可能になります。エイリアスが指定されなければ、サービス名で到達できます。

また、サービス間の依存関係は depends_on を使っても同様に指定できますし、サービスを起動する順番も指定できます。

links と networks を両方定義する時は、リンクするサービスが通信するために、ネットワークの少なくとも１つを共有する必要があります。

バージョン２のファイル形式のみ対応しています。バージョン１では log_driver と log_opt をお使いください。

サービスに対してログ記録の設定をします。

```
logging:
  driver: syslog
  options:
    syslog-address: "tcp://192.168.0.42:123"
```

driver にはコンテナのサービスに使うロギング・ドライバを指定します。これは docker run コマンドにおける --log-driver オプションと同じです。

デフォルトの値は json-file です。

```
driver: "json-file"
driver: "syslog"
driver: "none"
```

docker-compose up で立ち上げた場合、docker-compose logs コマンドでログを表示できるのは json-file ドライバを指定した時のみです。他のドライバを指定したら logs コマンドを実行しても画面に表示されません。

ロギング・ドライバのオプションを指定するには options キーを使います。これは docker run コマンド実行時の --log-opt オプションと同じです。

ロギングのオプションはキーバリューのペアです。以下は syslog オプションを指定する例です。

```
driver: "syslog"
options:
  syslog-address: "tcp://192.168.0.42:123"
```

log_driver

ファイル形式バージョン１のオプションです。バージョン２では logging を使います。

ログ・ドライバを指定します。デフォルトは json-file（JSON ファイル形式）です。

```
log_driver: "syslog"
```

log_opt

ファイル形式バージョン１のオプションです。バージョン２では logging を使います。

ログ記録のオプション、キー・バリューのペアで指定します。次の例は syslog のオプションです。

```
log_opt:
  syslog-address: "tcp://192.168.0.42:123"
```

net

ファイル形式バージョン１のオプションです。バージョン２では network_mode を使います。

ネットワーク・モードを指定します。これは docker クライアントで --net パラメータを指定するのと同じものです。コンテナ名や ID の代わりに、container:...で指定した名前が使えます。

```
net: "bridge"
net: "none"
net: "host"
net: "container:[サービス名かコンテナ名/id]"
```

network_mode

ファイル形式バージョン２のオプションです。バージョン１では net を使います。

ネットワーク・モードです。docker クライアントで --net パラメータを使うのと同じ働きですが、サービス:[サービス名] の形式で指定します。

```
network_mode: "bridge"
network_mode: "host"
network_mode: "none"
network_mode: "service:[service name]"
network_mode: "container:[container name/id]"
```

networks

ファイル形式バージョン 2 のオプションです。バージョン 1 では使えません。

ネットワークに参加する時、トップ・レベルの network キーのエントリを参照します。

```
services:
  some-service:
    networks:
     - some-network
     - other-network
```

aliases

エイリアス（ホスト名の別名）は、ネットワーク上のサービスに対してです。同一ネットワーク上の他のコンテナが、サービス名またはこのエイリアスを使い、サービスのコンテナの１つに接続します。

aliases が適用されるのはネットワーク範囲内のみです。そのため、同じサービスでも他のネットワークからは異なったエイリアスが使えます。

複数のコンテナだけでなく複数のサービスに対しても、ネットワーク範囲内でエイリアスが利用できます。ただしその場合、名前解決がどのコンテナに対して名前解決されるのか保証されません。

一般的な形式は、以下の通りです。

```
services:
  some-service:
    networks:
      some-network:
        aliases:
         - alias1
         - alias3
      other-network:
        aliases:
         - alias2
```

この例では、３つのサービス（web、worker、db）と２つのネットワーク（new と legacy）が提供されています。db サービスはホスト名 db または database として new ネットワーク上で到達可能です。そして、legacy ネットワーク上では db または mysql として到達できます。

```
version: '2'

services:
  web:
    build: ./web
    networks:
      - new

  worker:
    build: ./worker
    networks:
```

```
      - legacy

  db:
    image: mysql
    networks:
      new:
        aliases:
          - database
      legacy:
        aliases:
          - mysql

networks:
  new:
  legacy:
```

IPv4 アドレス、IPv6 アドレス

サービスをネットワークに追加する時、コンテナに対して静的な IP アドレスを割り当てます。

トップレベルのネットワーク・セクションでは、適切なネットワーク設定に ipam ブロックが必要です。ここで各静的アドレスが扱うサブネットやゲートウェイを定義します。IPv6 アドレスが必要であれば、com.docker.network.enable_ipv6 ドライバ・オプションを true にする必要があります。

例：

```
version: '2'

services:
  app:
    image: busybox
    command: ifconfig
    networks:
      app_net:
        ipv4_address: 172.16.238.10
        ipv6_address: 2001:3984:3989::10

networks:
  app_net:
    driver: bridge
    driver_opts:
      com.docker.network.enable_ipv6: "true"
    ipam:
      driver: default
      config:
      - subnet: 172.16.238.0/24
        gateway: 172.16.238.1
      - subnet: 2001:3984:3989::/64
        gateway: 2001:3984:3989::1
```

pid

```
pid: "host"
```

PID モードはホストの PID モードを設定します。有効化したら、コンテナとホスト・オペレーティング・シス

テム間で PID アドレス空間を共有します。コンテナにこのフラグを付けて起動したら、他のコンテナからアクセスできるだけでなく、ベアメタル・マシン上の名前空間などから操作できるようになります。

ports

公開用のポートです。ホスト側とコンテナ側の両方のポートを指定（ ホスト側:コンテナ側 ）できるだけでなく、コンテナ側のポートのみも指定できます（ホスト側はランダムなポートが選ばれます)。

ホスト側:コンテナ側 の書式でポートを割り当てる時、コンテナのポートが 60 以下であればエラーが発生します。これは YAML が xx:yy 形式の指定を、60 進数（60 が基準）の数値とみなすからです。そのため、ポートの割り当てには常に文字列として指定することを推奨します（訳者注：" で囲んで文字扱いにする)。

```
ports:
 - "3000"
 - "3000-3005"
 - "8000:8000"
 - "9090-9091:8080-8081"
 - "49100:22"
 - "127.0.0.1:8001:8001"
 - "127.0.0.1:5000-5010:5000-5010"
```

security_opt

各コンテナに対するデフォルトのラベリング・スキーマ（labeling scheme）を上書きします。

```
security_opt:
  - label:user:USER
  - label:role:ROLE
```

stop_signal

コンテナに対して別の停止シグナルを設定します。デフォルトでは stop で SIGTERM を使います。stop_signal で別のシグナルを指定したら、stop 実行時にそのシグナルを送信します。

```
stop_signal: SIGUSR1
```

ulimits

コンテナのデフォルト ulimits を上書きします。単一の整数値で上限を指定できるだけでなく、ソフト／ハード・リミットの両方も指定できます。

```
ulimits:
  nproc: 65535
  nofile:
    soft: 20000
    hard: 40000
```

volumes, volume_driver

マウント・パスまたは名前を付けたボリュームは、オプションでホストマシン（ ホスト:コンテナ ）上のパス指定や、アクセス・モード（ ホスト:コンテナ:rw ） を指定できます。 バージョン２のファイル では名前を付けたボリュームを使うにはトップ・レベルの volumes キーを指定する必要があります。バージョン１の場合は、ボリ

ュームが存在していなければ Docker Engine が自動的に作成します。

ホスト上の相対パスをマウント可能です。相対パスは Compose 設定ファイルが使っているディレクトリを基準とします。相対パスは . または .. で始まります。

```
volumes:
  # パスを指定したら、Engine はボリュームを作成
  - /var/lib/mysql

  # 絶対パスを指定しての割り当て
  - /opt/data:/var/lib/mysql

  # ホスト上のパスを指定する時、Compose ファイルからのパスを指定
  - ./cache:/tmp/cache

  # ユーザの相対パスを使用
  - ~/configs:/etc/configs/:ro

  # 名前付きボリューム（Named volume）
  - datavolume:/var/lib/mysql
```

ホスト側のパスを指定せず、volume_driver を指定したい場合があるかもしれません。

```
volume_driver: mydriver
```

バージョン２のファイルでは、名前付きボリュームに対してドライバを適用できません（ボリュームを宣言するのではなく、driver オプションを使ったほうが良いでしょう)。バージョン１の場合は、ドライバを指定すると名前付きボリュームにもコンテナのボリュームにも適用されます。

volume_driver も指定しても、パスは拡張されません。

詳しい情報は Docker ボリュームとボリューム・プラグインのドキュメントをご覧ください。

volumes_from

他のサービスやコンテナ上のボリュームをマウントします。オプションで、読み込み専用のアクセス（ro）や読み書き（rw）を指定できます。

```
volumes_from:
 - service_name
 - service_name:ro
 - container:container_name
 - container:container_name:rw
```

コンテナ:...　の形式をサポートしているのはバージョン２のファイル形式のみです。バージョン１の場合は、次のように明示しなくてもコンテナ名を使えます。

- service_name
- service_name:ro
- container_name
- container_name:rw

その他

cpu_shares、cpuset、domainname、entrypoint、hostname、ipc、mac_address、mem_limit、memswap_limit、privileged、read_only、restart、stdin_open、tty、user、working_dir は、それぞれ単一の値を持ちます。いずれも docker run コマンドのオプションに対応しています。

```
cpu_shares: 73
cpu_quota: 50000
cpuset: 0,1

user: postgresql
working_dir: /code

domainname: foo.com
hostname: foo
ipc: host
mac_address: 02:42:ac:11:65:43

mem_limit: 1000000000
memswap_limit: 2000000000
privileged: true

restart: always

read_only: true
stdin_open: true
tty: true
```

5.1.2 ボリューム設定リファレンス

サービス宣言の一部として、オン・ザ・フライでボリュームを宣言できます。このセクションでは名前付きボリューム（named volume）の作成方法を紹介します。このボリュームは複数のサービスを横断して再利用可能なものです（`volumes_from` に依存しません）。そして docker コマンドラインや API を使って、簡単に読み込みや調査が可能です。docker volumes のサブコマンドの詳細から、詳しい情報をご覧ください。

driver

ボリューム・ドライバがどのボリュームを使うべきかを指定します。デフォルトは `local` です。ドライバを指定しなければ、Docker Engine はエラーを返します。

```
driver: foobar
```

driver_opts

ボリュームが使うドライバに対して、オプションをキーバリューのペアで指定します。これらのオプションはドライバに依存します。オプションの詳細については、各ドライバのドキュメントをご確認ください。

```
driver_opts:
  foo: "bar"
  baz: 1
```

external

このオプションを true に設定したら、Compose の外にあるボリュームを作成します（訳者注：Compose が管理していない Docker ボリュームを利用します、という意味）。docker-compose up を実行してもボリュームを作成しません。もしボリュームが存在していなければ、エラーを返します。

external は他のボリューム用の設定キー（driver、driver_opts）と一緒に使えません。

以下の例は、［プロジェクト名]_data という名称のボリュームを作成する代わりに、Compose は data という名前で外部に存在するボリュームを探し出し、それを db サービスのコンテナの中にマウントします。

```
version: '2'

services:
  db:
    image: postgres
    volumes:
      - data:/var/lib/postgres/data

volumes:
  data:
    external: true
```

また、Compose ファイルの中で使われている名前を参照し、ボリューム名を指定可能です。

```
volumes
  data:
    external:
      name: actual-name-of-volume（実際のボリューム名）
```

5.1.3 ネットワーク設定リファレンス

ネットワークを作成するには、トップレベルの networks キーを使って指定します。Compose 上でネットワーク機能を使うための詳細情報は、Compose のネットワーク機能のセクションをご覧ください。

driver

対象のネットワークが使用するドライバを指定します。

デフォルトでどのドライバを使用するかは Docker Engine の設定に依存します。一般的には単一ホスト上であれば bridge でしょうし、Swarm 上であれば overlay でしょう。

ドライバが使えなければ、Docker Engine はエラーを返します。

```
driver: overlay
```

driver_opts

ネットワークが使うドライバに対して、オプションをキーバリューのペアで指定します。これらのオプションはドライバに依存します。オプションの詳細については、各ドライバのドキュメントをご確認ください。

```
driver_opts:
  foo: "bar"
  baz: 1
```

ipam

IPAM（IP アドレス管理）のカスタム設定を指定します。様々なプロパティ（設定）を持つオブジェクトですが、各々の指定はオプションです。

- driver：デフォルトの代わりに、カスタム IPAM ドライバを指定します。
- config：ゼロもしくは複数の設定ブロック一覧です。次のキーを使えます。
  - subnet：ネットワーク・セグメントにおける CIDR のサブネットを指定します。
  - ip_range：コンテナに割り当てる IP アドレスの範囲を割り当てます。
  - gateway：マスタ・サブネットに対する IPv4 または IPv6 ゲートウェイを指定します。
  - aux_addresses：ネットワーク・ドライバが補助で使う IPv4 または IPv6 アドレスを指定します。これはホスト名を IP アドレスに割り当てるためのものです。

全てを使った例：

```
ipam:
  driver: default
  config:
    - subnet: 172.28.0.0/16
      ip_range: 172.28.5.0/24
      gateway: 172.28.5.254
      aux_addresses:
        host1: 172.28.1.5
        host2: 172.28.1.6
        host3: 172.28.1.7
```

external

このオプションを true に設定したら、Compose の外にネットワークを作成します（訳者注：Compose が管理していない Docker ネットワークを利用します、という意味）。docker-compose up を実行してもネットワークを作成しません。もしネットワークが存在していなければ、エラーを返します。

external は他のネットワーク用の設定キー（driver、driver_opts、ipam）と一緒に使えません。

以下の例は、外の世界とのゲートウェイに proxy を使います。 [プロジェクト名]_outside という名称のネットワークを作成する代わりに、Compose は outside という名前で外部に存在するネットワークを探し出し、それを proxy サービスのコンテナに接続します。

```
version: '2'

services:
  proxy:
    build: ./proxy
    networks:
      - outside
      - default
  app:
    build: ./app
    networks:
      - default

networks:
  outside:
```

```
    external: true
```

また、Compose ファイルの中で使われている名前を参照し、ネットワーク名を指定可能です。

```
networks:
  outside:
    external:
      name: actual-name-of-network
```

バージョン

Compose ファイル形式には 2 つのバージョンがあります。

- バージョン 1 は過去のフォーマットです。YAML の冒頭で `version` キーを指定不要です。
- バージョン 2 は推奨フォーマットです。YAML の冒頭で `version: '2'`のエントリを指定します。

プロジェクトをバージョン 1 からバージョン 2 に移行する方法は、「アップグレード方法」のセクションをご覧ください。

複数の Compose ファイル や 拡張サービス を使う場合は、各ファイルが同じバージョンでなくてはいけません。 1 つのプロジェクト内でバージョン 1 と 2 を混在できません。

バージョンごとに異なった制約があります。

- 構造と利用可能な設定キー
- 実行に必要な Docker Engine の最低バージョン
- ネットワーク機能に関する Compose の挙動

これらの違いを、以下で説明します。

バージョン 1

Compose ファイルでバージョンを宣言しなければ「バージョン 1 」として考えます。バージョン 1 では、ドキュメントの冒頭から全てのサービスを定義します。
バージョン 1 は **Compose 1.6.x まで**サポートされます。今後の Compose バージョンでは廃止予定です。
バージョン 1 のファイルでは volumes、networks、build 引数を使えません。
例：

```
web:
  build: .
  ports:
   - "5000:5000"
  volumes:
   - .:/code
  links:
   - redis
redis:
  image: redis
```

バージョン２

バージョン２の Compose ファイルでは、ドキュメントの冒頭でバージョン番号を明示する必要があります。services キーの下で全てのサービスを定義する必要があります。

バージョン２のファイルは Compose 1.6.0 以上でサポートされており、実行には Docker Engine 1.10.0 以上が必要です。

名前付きボリュームの宣言は `volumes` キーの下で行えます。また、名前付きネットワークの宣言は `networks` キーの下で行えます。

シンプルな例：

```
version: '2'
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
  redis:
    image: redis
```

ボリュームとネットワークを定義するよう拡張した例：

```
version: '2'
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
    networks:
      - front-tier
      - back-tier
  redis:
    image: redis
    volumes:
      - redis-data:/var/lib/redis
    networks:
      - back-tier
volumes:
  redis-data:
    driver: local
networks:
  front-tier:
    driver: bridge
  back-tier:
    driver: bridge
```

アップグレード方法

ほとんどの場合、バージョン１から２への移行はとても簡単な手順です。

- 最上位レベルとして services: キーを追加する。
- ファイルの１行め冒頭に version: '2' を追加する。

特定の設定機能を使っている場合は、より複雑です。

- dockerfile：build キー配下に移動します。

```
build:
  context: .
  dockerfile: Dockerfile-alternate
```

- log_driver、log_opt：これらは logging キー以下です。

```
logging:
  driver: syslog
  options:
    syslog-address: "tcp://192.168.0.42:123"
```

- links と環境変数： 環境変数リファレンス に文章化している通り、links によって作成される環境変数機能は、いずれ廃止予定です。新しい Docker ネットワーク・システム上では、これらは削除されています。ホスト名のリンクを使う場合は、適切なホスト名で接続できるように設定するか、あるいは自分自身で代替となる環境変数を指定します。

```
web:
  links:
    - db
  environment:
    - DB_PORT=tcp://db:5432
```

- external_links：バージョン２のプロジェクトを実行する時、Compose は Docker ネットワーク機能を使います。つまり、これまでのリンク機能と挙動が変わります。典型的なのは、２つのコンテナが通信するためには、少なくとも１つのネットワークを共有する必要があります。これはリンク機能を使う場合でもです。

外部のコンテナがアプリケーションのデフォルト・ネットワークに接続する場合や、自分で作成したサービスが外部のコンテナと接続するには、外部ネットワーク機能を使います。

- net：これは network_mode に置き換えられました。

```
net: host    ->  network_mode: host
net: bridge  ->  network_mode: bridge
net: none    ->  network_mode: none
```

net: "コンテナ:[サービス名]" を使っていた場合は、network_mode: "サービス:[サービス名]" に置き換える必要があります。

```
net: "container:web"  ->  network_mode: "service:web"
```

net: "コンテナ:[コンテナ名/ID]" の場合は変更不要です。

```
net: "container:cont-name"  ->  network_mode: "container:cont-name"
```

```
net: "container:abc12345"   ->  network_mode: "container:abc12345"

net: “container:abc12345” -> network_mode: “container:abc12345”
```

- volumes を使う名前付きボリューム：Compose ファイル上で、トップレベルの volumes セクションとして明示する必要があります。data という名称のボリュームにサービスがマウントする必要がある場合、トップレベルの volumes セクションで data ボリュームを宣言する必要があります。記述は以下のような形式です。

```
version: '2'
services:
  db:
    image: postgres
    volumes:
      - data:/var/lib/postgresql/data
volumes:
  data: {}
```

デフォルトでは、Compose はプロジェクト名を冒頭に付けたボリュームを作成します。data のように名前を指定するには、以下のように宣言します。

```
volumes:
  data:
    external: true
```

変数の置き換え

設定オプションでは環境変数も含めることができます。シェル上の Compose は docker-compose の実行時に環境変数を使えます。たとえば、シェルで EXTERNAL_PORT=8000 という変数を設定ファイルで扱うには、次のようにします。

```
web:
  build: .
  ports:
    - "${EXTERNAL_PORT}:5000"
```

この設定で docker-compose up を実行したら、Compose は EXTERNAL_PORT 環境変数をシェル上で探し、それを値と置き換えます。この例では、Compose が web コンテナを作成する前に“ 8000:5000 ”のポート割り当てをします。

環境変数が設定されていなければ、Compose は空の文字列に置き換えます。先の例では、EXTERNAL_PORT が設定されなければ、 ポートの割り当ては :5000　になります（もちろん、これは無効なポート割り当てなため、コンテナを作成しようとしてもエラーになります)。

$変数 と ${変数} の両方がサポートされています。シェルの拡張形式である $変数-default と ${変数/foo/bar} はサポートされません。

$$（二重ドル記号）を指定する時は、設定ファイル上でリテラルなドル記号の設定が必要です。Compose は値を補完しませんので、 $$の指定により、Compose によって処理されずに環境変数を参照します。

```
web:
  build: .
  command: "$$VAR_NOT_INTERPOLATED_BY_COMPOSE"
```

もしも間違えてドル記号（ $ ）だけにしたら、Compose は環境変数の値を解釈し、次のように警告を表示しま

す。

```
The VAR_NOT_INTERPOLATED_BY_COMPOSE is not set. Substituting an empty string.
```

5.2 docker-compose コマンド

このセクションは docker-compose サブコマンドに関する説明を扱っています。また、コマンドラインで docker-compose [サブコマンド] --help を実行しても情報を表示します。

5.2.1 docker-compose コマンド概要

このセクションは docker-compose コマンドの使い方に関する情報を扱います。この情報はコマンドライン上で docker-compose --help を使っても確認できます。

```
Docker で使う複数コンテナ・アプリケーションの定義と実行

使い方:
  docker-compose [-f=<引数>...] [オプション] [コマンド] [引数...]
  docker-compose -h|--help

オプション:
  -f, --file FILE             別の compose ファイルを指定（デフォルト: docker-compose.yml）
  -p, --project-name NAME     別のプロジェクト名を指定（デフォルト: directory name）
  --verbose                   詳細情報を表示
  -v, --version               バージョンを表示して終了
  -H, --host HOST             接続先のデーモン・ソケット

  --tls                       TLS を使う;--tlsverify の指定も含む
  --tlscacert CA_PATH         この CA で署名した証明書のみ信頼
  --tlscert CLIENT_CERT_PATH  TLS 証明書ファイルへのパス
  --tlskey TLS_KEY_PATH       TLS 鍵ファイルへのパス
  --tlsverify                 TLS を使いリモートを認証
  --skip-hostname-check       クライアントの証明書で指定されたデーモンのホスト名を確認しない。
                              （たとえば、docker ホストが IP アドレスの場合）

コマンド:
  build              サービスの構築または再構築
  config             compose ファイルの確認と表示
  create             サービスの作成
  down               コンテナ・ネットワーク・イメージ・ボリュームの停止と削除
  events             コンテナからリアルタイムにイベントを受信
  help               コマンド上でヘルプを表示
  kill               コンテナを kill（強制停止）
  logs               コンテナの出力を表示
  pause              サービスを一時停止
  port               ポートに割り当てる公開用ポートを表示
  ps                 コンテナ一覧
  pull               サービス用イメージの取得
  restart            サービスの再起動
  rm                 停止中のコンテナを削除
  run                １度だけコマンドを実行
  scale              サービス用コンテナの数を指定
  start              サービスの開始
  stop               サービスの停止
  unpause            サービスの再開
  up                 コンテナの作成と開始
  version            Docker Compose のバージョン情報を表示
```

docker-compose は Docker Compose のバイナリです。このコマンドを使い Docker コンテナ上の複数のサービスを管理します。

Compose 設定ファイルの場所を指定するには、 -f フラグを使います。複数の -f 設定ファイルを指定できます。複数のファイルを指定したら、Compose は 1 つの設定ファイルに連結します。Compose はファイルを指定した順番で構築します。後に続くファイルは、既に実行したものを上書き・追加します。

たとえば、次のようなコマンドラインを考えます。

```
$ docker-compose -f docker-compose.yml -f docker-compose.admin.yml run backup_db`
```

docker-compose.yml ファイルは webapp サービスを指定しています。

```
webapp:
  image: examples/web
  ports:
    - "8000:8000"
  volumes:
    - "/data"
```

また、docker-compose.admin.yml ファイルで同じサービスを指定したら、以前のファイルで指定した同じフィールドの項目があれば、それを上書きします。新しい値があれば、webapp サービスの設定に追加します。

```
webapp:
  build: .
  environment:
    - DEBUG=1
```

-f に -（ダッシュ）をファイル名として指定したら、標準入力から設定を読み込みます。設定に標準入力を使う場合のパスは、現在の作業用ディレクトリからの相対パスとなります。

-f フラグはオプションです。コマンドラインでこのフラグを指定しなければ、Compose は現在の作業用ディレクトリと docker-compose.yml ファイルと docker-compose.override.yml ファイルのサブディレクトリを探します。もし、２つのファイルを指定したら、１つの設定ファイルに連結します。この時、docker-compose.yml ファイルにある値は、docker-compose.override.yml ファイルで設定した値で上書きします。

詳しくは次の「COMPOSE 環境変数」セクションをご覧ください。

各設定ファイルはプロジェクト名を持っています。 -p フラグでプロジェクト名を指定できます。フラグを指定しなければ、Compose は現在のディレクトリの名前を使います。詳細は「COMPOSE_PROJECT 環境変数」のセクションをご覧ください。

5.2.2 CLI 環境変数

Docker Compose のコマンドラインでの動作を設定するために、複数の環境変数を利用可能です。

DOCKER_ で始まる環境変数は、Docker コマンドライン・クライアントで用いられている設定と同じです。もしも docker-machine を使っているのであれば、eval "$(docker-machine env my-docker-vm)" コマンドで適切な環境変数の値を設定します（この例では、my-docker-vm は Docker Machine で作成したマシンの名前です ）。

環境ファイル(.env)を使っても変数を指定できます。

COMPOSE_PROJECT_NAME

プロジェクト名を設定します。この値はコンテナの起動時に、コンテナのサービス名の先頭に付けられます。た

とえば、2 つのサービス db と web を持つプロジェクトの名前を myapp としたら、Compose は myapp_db_1 と myapp_web_1 と名前の付いたコンテナをそれぞれ起動します。

このプロジェクト名の設定はオプションです。設定をしなければ、COMPOSE_PROJECT_NAME（Compose のプロジェクト名）は、デフォルトではプロジェクトのディレクトリをベース名にします。詳しくはコマンドライン・オプション -p をご覧ください。

COMPOSE_FILE

Compose 設定を含むファイルを指定します。指定しなければ、Compose は現在のディレクトリにある docker-compose.yml という名称のファイルを探します。あるいは、親ディレクトリにあれば、そちらを使います。詳しくはコマンドライン・オプション -f をご覧ください。

COMPOSE_API_VERSION

Docker API は、明確なバージョンを報告するクライアントに対してのみ応答します。docker-compose を使う時、client and server don't have same version error というエラーが出る場合は、このエラーを回避するために環境変数を設定します。バージョンの値がサーバのバージョンと一致するように設定します。

この変数を設定するのは、クライアントとサーバのバージョンが一致しない場合でも、一時的に回避してコマンドを実行したい場合です。たとえば、クライアントをアップグレードしていても、サーバのアップグレードまで待つ必要がある場合です。

この環境変数を設定したら、いくつかの Docker の機能が正常に動作しない可能性があります。実際にどのような挙動になるかは、クライアントとサーバのバージョンによって変わります。そのため、この環境変数を設定するのは、あくまで回避策であって、公式にサポートされている手法ではありません。

もしこの環境変数を設定して何か問題が起きた場合は、サポートに解決策を訊ねる前に、バージョンの差違を解消した後、環境変数を削除してください。

DOCKER_HOST

docker デーモンの URL を設定します。Docker クライアントのデフォルトは unix:///var/run/docker.sock です。

DOCKER_TLS_VERIFY

空白以外の何らかの値をセットしたら、docker デーモンとの TLS 通信を有効化します。

DOCKER_CERT_PATH

TLS 認証に使う ca.pem、cert.pem、key.pem ファイルのパスを設定します。デフォルトは ~/.docker です。

COMPOSE_HTTP_TIMEOUT

Compose が Docker デーモンに対する処理が失敗（fail）したとみなす時間（秒単位）を設定します。デフォルトは 60 秒です。

5.3 Compose コマンドライン・リファレンス

build

```
使い方: build [オプション] [サービス...]

オプション:
--force-rm  常に中間コンテナを削除
--no-cache  構築時にイメージのキャッシュを使わない
--pull      常に新しいバージョンのイメージ取得を試みる
```

サービスは プロジェクト名_サービス として構築時にタグ付けられます。例：composetest_db。サービスの Dockerfile や構築ディレクトリの内容に変更を加える場合は、docker-compose build で再構築を実行します。

config

```
使い方: config [オプション]

オプション:
-q, --quiet     認証設定以外のメッセージを表示しない
--services      サービス名を１行で表示
```

Compose ファイルを確認・表示します。

create

```
使い方: create [オプション] [サービス...]

オプション:
    --force-recreate       設定やイメージに変更がなくてもコンテナを再作成
                           --no-recreate とは同時に使えない
    --no-recreate          コンテナが存在していたら、再作成しない
                           --force-recreate とは同時に使えない
    --no-build             イメージがなくても構築しない
    --build                コンテナを作成前にイメージを作成
```

サービス用のコンテナを作成します。

down

コンテナを停止し、up で作成したコンテナ・ネットワーク・ボリューム・イメージを削除します。デフォルトではコンテナとネットワークのみ削除します。

```
使い方: down [オプション]

オプション:
    --rmi type           イメージの削除。type は次のいずれか:
                          'all': あらゆるサービスで使う全イメージを削除
                          'local': image フィールドにカスタム・タグのないイメージだけ削除
    -v, --volumes        Compose ファイルの `volumes` セクションの名前付きボリュームを削除
                          また、コンテナがアタッチしたアノニマス・ボリュームも削除
    --remove-orphans     Compose ファイルで定義していないサービス用のコンテナも削除
```

コンテナを停止し、 up で作成しコンテナ、ネットワーク、ボリューム、イメージを削除します。

デフォルトでは以下のものだけ削除します。

- Compose ファイル内で定義したサービス用のコンテナ
- Compose ファイルの network セクションで定義したネットワーク
- default ネットワーク（を使っている場合）

`external` として定義したネットワークとボリュームは決して削除しません。

events

```
使い方: events [オプション] [サービス...]

オプション:
    --json      json オブジェクトでイベントの出力をストリーム
```

プロジェクト内の各コンテナのイベントを表示します。
`--json` フラグを使うと、各行を JSON オブジェクトとして表示します。

```
{
    "service": "web",
    "event": "create",
    "container": "213cf75fc39a",
    "image": "alpine:edge",
    "time": "2015-11-20T18:01:03.615550",
}
```

help

```
使い方: help コマンド
```

コマンドのヘルプと使い方の指示を表示します。

kill

```
使い方: kill [オプション] [サービス...]

オプション:
-s SIGNAL         コンテナに送信するシグナル。デフォルトのシグナルは SIGKILL
```

`SIGKILL` シグナルを送信し、実行中のコンテナを強制停止します。次のように、オプションでシグナルを渡すこともできます。

```
$ docker-compose kill -s SIGINT
```

logs

```
使い方: logs [オプション] [サービス...]

オプション:
--no-color          白黒で画面に出力
-f, --follow        ログの出力をフォロー（表示し続ける）
-t, --timestamps    タイムスタンプの表示
--tail              各コンテナのログの最終行から遡った行を表示
```

サービスからのログ出力を表示します。

pause

```
使い方: pause [サービス...]
```

サービスを実行しているコンテナを一時停止（pause）します。再開（unpause）するには `docker-compose unpause` を使います。

port

```
使い方: port [オプション] サービス プライベート_ポート

オプション:
--protocol=proto  tcp or udp [デフォルトj : tcp]
--index=index     サービスに複数のインスタンスがある場合、コンテナのインデックス数 [デフォルト: 1]
```

ポートを割り当てる公開用のポートを表示します。

ps

```
使い方: ps [オプション] [サービス...]

オプション:
-q    ID のみ表示
```

コンテナ一覧を表示します。

pull

使い方: pull [オプション] [サービス...]

オプション:

--ignore-pull-failures　取得に失敗しても無視する

```
サービス用イメージを取得します。
```

restart

```
使い方: restart [オプション] [サービス...]

Options:
-t, --timeout TIMEOUT      シャットダウンのタイムアウト秒数を指定（デフォルト: 10）
```

サービスを再起動します。

rm

```
使い方: rm [オプション] [サービス...]

オプション:
    -f, --force   確認なく削除する
    -v            コンテナにアタッチしているアノニマス・ボリュームも削除
    -a, --all     docker-compose run で作成した一度だけのコンテナを全て削除
                  docker-compose run
```

停止済みのサービス・コンテナを削除します。

デフォルトでは、コンテナにアタッチしている匿名ボリューム（anonymous volume）を削除しません。ボリュームを削除するには -v オプションを使います。全てのボリュームを表示するには docker volume ls を使います。

（明示的に削除しなければ）ボリューム内にあるデータは失われません。

run

```
使い方: run [オプション] [-e キー=バリュー...] サービス [コマンド] [引数...]

オプション:
-d                    デタッチド・モード: コンテナをバックグラウンドで実行し、新しいコンテナ名を表示
--name NAME           コンテナに名前を割り当て
--entrypoint CMD      イメージのエントリ・ポイントを上書き
-e KEY=VAL            環境変数を指定（複数回指定できる）
-u, --user=""         実行時のユーザ名または uid を指定
--no-deps             リンクしたサービスを起動しない
--rm                  コンテナ実行後に削除お。デタッチド・モードの場合は無視
-p, --publish=[]      コンテナのポートをホスト側に公開
--service-ports       サービス用のポートを有効化し、ホスト側に割り当て可能にする
-T                    疑似ターミナル（pseudo-tty）割り当てを無効化
                      デフォルトの `docker-compose run` は TTY を割り当て
-w, --workdir=""      コンテナ内のワーキング・ディレクトリを指定
```

サービスに対して１回コマンドを実行します。たとえば、次のコマンドは web サービスを開始するためのコマンドで、サービス内で bash としてコマンドを実行します。

```
$ docker-compose run web bash
```

run コマンドを使うと、サービスの設定ファイルで定義された通りに、同じ設定の新しいコンテナを開始します。つまり、コンテナは設定ファイル上で定義された同じボリュームとリンクを持ちます。ただ、ここでは２つの違いがあります。

１つは、run コマンドの指定は、サービス設定ファイル上での定義を上書きします。たとえば、web サービスは bash で開始する設定だとしても、docker-compose run web python app.py を実行すると、python app.py で上書きします。

２つめの違いとして、docker-compose run コマンドはサービス設定ファイルで指定したポートを作成しません。これは、既に開いているポートとの衝突を避けるためです。サービス用のポートを作成し、ホスト側に割り当てるには、--service-ports フラグを使います。

```
$ docker-compose run --service-ports web python manage.py shell
```

別の方法として、手動でポートの割り当てを設定することも可能です。同様に Docker で run コマンドを使う時に、--publish または -p オプションを使います。

```
$ docker-compose run --publish 8080:80 -p 2022:22 -p 127.0.0.1:2021:21 web python manage.py shell
```

リンク機能を使ってサービスを開始する場合、run コマンドはリンク先のサービスが実行中かどうかをまず確認し、サービスが停止していれば起動します。全てのリンク先のサービスが起動したら、指定したコマンドで run 命令が実行されます。たとえば、次のように実行できます。

```
$ docker-compose run db psql -h db -U docker
```

これはリンクしている db コンテナに対して、PostgreSQL シェルで操作をします。

run コマンドを実行する時、リンクしているコンテナを起動したくない場合は --no-deps フラグを使います。

```
$ docker-compose run --no-deps web python manage.py shell
```

scale

```
使い方: scale [サービス=数値...]
```

サービスを実行するコンテナ数を設定します。
数は service=数値 の引数で指定します。実行例：

```
$ docker-compose scale web=2 worker=3
```

start

```
使い方: start [サービス...]
```

既存のコンテナをサービスとして起動します

stop

```
使い方: stop [オプション] [サービス...]

オプション:
-t, --timeout TIMEOUT      シャットダウンのタイムアウト秒数を指定（デフォルト: 10).
```

稼働中のコンテナを停止しますが、削除しません。 docker-compose start コマンドで、再起動できます。

unpause

```
使い方: unpause [サービス...]
```

停止中（paused）のサービスを再開します。

up

```
使い方: up [オプション] [サービス...]

オプション:
    -d                         デタッチド・モード: バックグラウンドでコンテナを実行し、新しいコンテナ
名を表示
                               --abort-on-container-exit と同時に使えない
    --no-color                 白黒で画面に表示
    --no-deps                  リンクしたサービスを表示しない
    --force-recreate           設定やイメージに変更がなくても、コンテナを再作成する
                               --no-recreate と同時に使えません
    --no-recreate              コンテナが既に存在していれば、再作成しない
                               --force-recreate と同時に使えない
    --no-build                 イメージが見つからなくても構築しない
    --build                    コンテナを開始前にイメージを構築する
    --abort-on-container-exit  コンテナが１つでも停止したら全てのコンテナを停止
                               -d と同時に使えない
    -t, --timeout TIMEOUT      アタッチしている、あるいは既に実行中のコンテナを
                               停止する時のタイムアウト秒数を指定（デフォルト:10 ）
```

`--remove-orphans`	Compose ファイルで定義されていないサービス用のコンテナを削除

サービス用のコンテナの構築、作成、起動、アタッチを行います。

既に実行している場合は、このコマンドによってリンクされているサービスも起動します。

`docker-compose up` コマンドは各コンテナの出力を統合します。コマンドを終了（exit）すると、全てのコンテナを停止します。`docker-compose up -d` で実行すると、コンテナをバックグラウンドで起動し、実行し続けます。

もしサービス用のコンテナが存在している場合、かつ、コンテナを作成後にサービスの設定やイメージを変更している場合は、`docker-compose up -d` を実行すると、 設定を反映するためにコンテナを停止・再作成します（マウントしているボリュームは、そのまま保持します）。Compose が設定を反映させないようにするには、`--no-cecreate` フラグを使います。

もしも Compose で処理時、強制的に全てのコンテナを停止・再作成するには `--force-recreate` フラグを使います。

5.4 リンク環境変数リファレンス

サービスをリンクで接続する手法としては、環境変数の使用は推奨されなくなりました。その代わりに、接続するホスト名として、名前を使ったリンクが可能です（デフォルトではサービスの名前でリンクします)。詳細は docker-compose.yml ドキュメントをご覧ください。

環境変数を（Compose で自動的に扱えるように）は、 過去の Compose ファイル形式バージョン１を使う場合のみです。

Compose はサービスのコンテナを他に公開するために、Docker リンク機能を使います。リンクされた各コンテナは環境変数のセットを持ます。環境変数は各コンテナ名を大文字にしたもので始まります。

どのような環境変数が設定されているかを確認するには、`docker-compose run サービス名 env` を実行します。

name_PORT

全ての URL です。例：`DB_PORT=tcp://172.17.0.5:5432`

name_PORT_num_protocol

全ての URL です。例：`DB_PORT_5432_TCP=tcp://172.17.0.5:5432`

name_PORT_num_protocol_ADDR

コンテナの IP アドレスです。例：`DB_PORT_5432_TCP_ADDR=172.17.0.5`

name_PORT_num_protocol_PORT

公開するポート番号です。例：`DB_PORT_5432_TCP_PORT=5432`

name_PORT_num_protocol_PROTO

プロトコル（tcp か udp）を指定します。例：`DB_PORT_5432_TCP_PROTO=tcp`

name_NAME

コンテナに対してコンテナ名としての完全修飾名（fully qualified container name）を指定します。例：`DB_1_NAME=/myapp_web_1/myapp_db_1`