SECTION 03

Dockerを体験してみよう

Dockerのインストールが完了したので、Dockerコンテナを展開することで概要を学びます。最初にhello-worldコンテナやnginxコンテナの展開を通してどのようにコンテナが作成されるか学び、実際にアプリを動かしてみます。そのあとでコンテナを作るイメージを管理するレジストリの使い方を説明します。

◎ Hello Worldイメージの展開

Dockerをはじめて利用する方のために「hello-world」というイメージが用意されているので、それを起動してみましょう。コンソールを開いて以下のコマンドを入力してください。コマンドの詳細は後述しますが、**dockerコマンド**に続けてcontainer（コンテナ）をrun（実行）するとし、その対象イメージとしてhello-worldを指定しています。

図1-30 hello-worldイメージからコンテナを作成

```
$ docker container run hello-world
中略

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

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.
後略
```

入力するとさまざまなメッセージが表示されたあとで、最終的に「Hello from Docker!」に続けて「このコンテナがどのように起動されたか」を説明するメッセージが出力されます。日本語で要約すると以下の通りです。

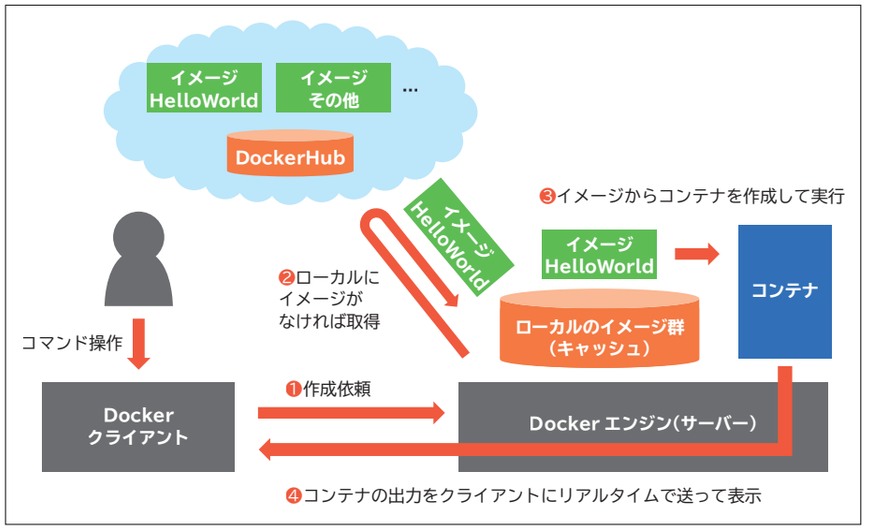
図1-31 日本語要約

```
このメッセージが見えたということは、インストールは成功したようです。
このメッセージを表示するために、Dockerは以下のことをやっています。

1. Dockerクライアント（dockerコマンド）がDockerデーモン（サーバー）に命令
2. Dockerデーモンが「hello-world」イメージをDockerHubより取得
3. Dockerデーモンがイメージからコンテナを作成し、メッセージ表示をするプログラムを実行
4. DockerデーモンがメッセージをDockerクライアントに送信し、それがターミナルに表示された
```

はじめての方には日本語にしてもわかりにくいかもしれませんので、仕組みを図にまとめました。

図1-32 コンテナが起動されるまでの流れ



Dockerの実行環境はサーバー上で「**Dockerエンジン（サーバープロセス）**」として動いています。詳しくは3章の後半で説明しますが、メッセージに表記される「**Dockerデーモン**」はDockerエンジンの一部であり、外部からの命令をREST API（HTTPベースのAPI規格）で受け付けます。そしてDockerデーモンは、Dockerエンジンの他のコンポーネントに実行を依頼します。先ほど利用したdockerコマンドは、このDockerエンジンに命令を投げるクライアントとして使われており、「hello-worldイメージのコンテナを走らせろ」と依頼しています。

クライアントから命令を受けたDockerエンジンは**イメージをDockerHubからダウンロード**してき

SECTION 01

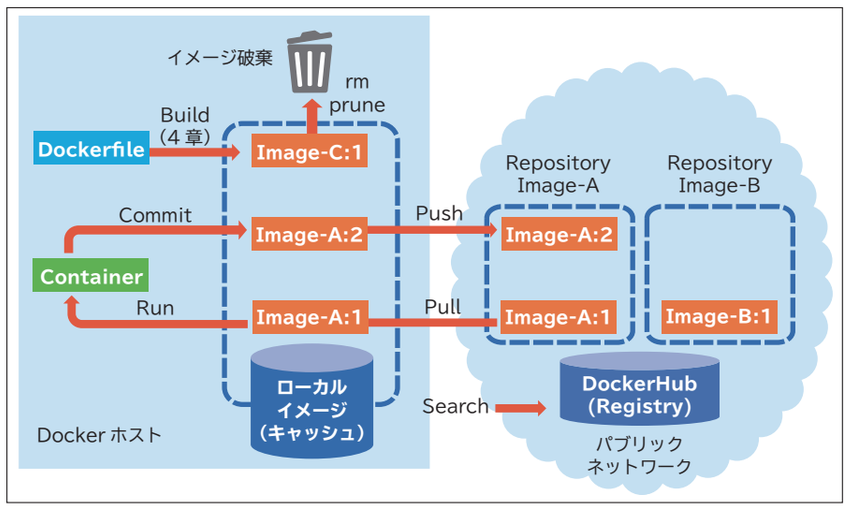
Dockerイメージを使いこなそう

Dockerはコンテナとしてアプリを動かしますが、そのコンテナのベースとなるのがイメージです。イメージをどのように扱うか把握していないと、正しくコンテナを作成することができません。本節ではレジストリ上のイメージの検索方法と、イメージのライフサイクルや利用法全般について扱います。

◎ イメージの使い方の全体像

イメージの全体像を把握するために、Dockerにおけるイメージの利用法を以下の図にまとめます。本章と4章（Dockerfile）を学ぶことで図にある操作をひと通り実施できるようになります。

図2-1 イメージの状態遷移図



まず、コンテナとして展開したり自分の独自イメージの開発をするためには、ベースとなるイメージを取得する必要があります。普通は0からイメージを作ることはしませんが、その気になればUbuntuを自作したり、4章で扱うバイナリのみ（OSがない）のイメージの作成もできます。

ベースとするイメージを見つけるには、dockerの検索機能やWeb検索などを利用します。見つけたイメージを取得する方法は**Pull（手動ダウンロード）と、自動取得（必要になった時点でDockerエンジンが自動で取得）**があります。そしてダウンロードされたイメージをrunコマンドなどで展開し、コンテナとして利用します。展開は何度でもできますので、コンテナ化してもイメージはなくなったりしません。

自分で独自イメージを作成する際は、**ダウンロードしたイメージに対して4章で扱うDockerfileで変更を加える**のが一般的な方法です。ただし、本章の後半では起動したコンテナに対して手動で変更を加えて、それをcommitコマンドでイメージ化するという方法を学びます。本章の内容は推奨されるイメージ作成方法ではありませんが、Dockerfileの仕組みを理解するための勉強になるので試してみてください。

◎ イメージを検索する

DockerHubには公式リポジトリや個人のリポジトリなど、さまざまなイメージが登録されています。dockerコマンドには「**docker search**」というレジストリを検索するための機能が備わっていますので、それを使って本章で利用するpythonのイメージを検索してみます。

図2-2 docker searchコマンドで検索

```
$ docker search python
NAME              DESCRIPTION                                     STARS  OFFICIAL  AUTOMATED
python            Python is an interpreted, interactive, objec…   4648   [OK]
django            Django is a free web application framework, …   894    [OK]
pypy              PyPy is a fast, compliant alternative implem…   216    [OK]
kaggle/python     Docker image for Python scripts run on Kaggle   129               [OK]
arm32v7/python    Python is an interpreted, interactive, objec…   42
後略
```

searchに続けて検索キーワードを指定すると、デフォルトでSTARS（人気）順でリポジトリが表示されます。OFFICIALという項目は公式リポジトリであることを意味しており、その横のAUTOMATEDは「Automated Build」と呼ばれる機能を使ってビルドされていることを示します。

searchコマンドの検索対象は**イメージ名だけでなくDESCRIPTIONも含まれます**。そのため、pythonという名前が付いたイメージに加えて、pythonのフレームワークであるdjangoリポジトリ（名前にpythonは含まない）も表示されています。searchにオプション「--no-trunc」を加えると、上記で「...」と省略されているDESCRIPTIONをすべて表示するので確認できます。

SECTION 04

Ansibleを使って Dockerホストを構築しよう

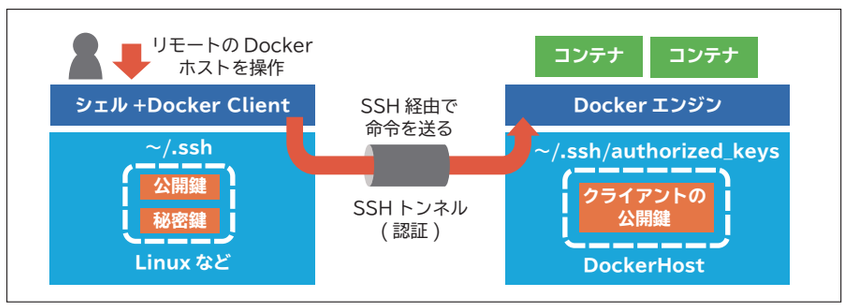
ここまではDocker Desktopもしくは、Dockerホスト上から自分自身に対しての操作を解説してきました。ここでは複数のDockerホストを用意する方法を学びます。Ansibleを利用すると、複数台のDockerホストでも自動的に構築できます。

リモートのDockerホストを使う

これまでのdockerの利用はDocker Desktopもしくは、Dockerホスト上から自分自身に対しての操作となりました。つまりDockerクライアント（dockerコマンド）が、同じホストにいるDockerエンジン（サーバーサイド）に対して命令を出して操作をしていたということです。これとほとんど同じようにホストA上のDockerクライアントが、ホストBのDockerエンジンをネットワーク越しに操作することができます。つまり**dockerコマンドを別のサーバーに対して実施する**ということです。

dockerコマンドを使ってリモートのdockerホストを操作するには、**SSH**を利用するのが一般的です。docker自体のセキュリティの仕組みではなく、SSHのセキュリティ（認証）を使うことで、決められたユーザーのみが外部からDockerホストを操作することができます。DockerのSSH利用は鍵（公開鍵暗号）を使った認証を利用します。適切に接続元／先で鍵の設定をすれば、リモートへの接続をパスワードなしで行えるようになります。以下に必要な設定を図にまとめます。

図2-33 リモートのDockerホストの利用



公開鍵と秘密鍵はペアとなる鍵で、名前の通り公開鍵は外部に公開可能で、秘密鍵はマシン内で大切に保管します。仕組みを説明すると長くなるため割愛しますが、接続されるホストは接続元の公開鍵（id_rsa.pub）をSSHフォルダの「authorized_keys」に登録しておくと、対応する秘密鍵（id_rsa）を持つホストからの接続はパスワードなしで行えるようになります。

SSHの鍵を登録する

ここではLinux-A（Dockerクライアント）のrootユーザーが、Linux-B（Dockerホスト）にrootユーザーとしてに接続するというシナリオで設定を行います。Hyper-VまたはVirtualBoxで**CentOS7をインストールした仮想マシンを3つ作成（複製）**してから読み進めてください（P.26参照）。

最初にLinux-Aで鍵を生成します。生成するファイル名が聞かれますが、デフォルトのパスに鍵を作成しますのでそのままEnterキーを押してください。

図2-34 Dockerクライアント側で鍵を生成

```
# ssh-keygen -t rsa -m PEM -q -N ""
Enter file in which to save the key (/root/.ssh/id_rsa):

# ls ~/.ssh/
id_rsa  id_rsa.pub  known_hosts
```

次に接続先のホスト（Linux-B）に対して鍵の登録を行います。手動で登録すると面倒なのでLinux-AからLinux-Bに対して「ssh-copy-id」コマンドで設定します。パスワード認証にパスすれば、自分の公開鍵が相手のauthorized_keysに登録されて、次回からは鍵認証（パスワードなし）でログインできるようになります。

図2-35 Dockerホストに鍵を登録

```
# ssh-copy-id root@10.149.245.208
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/root/.ssh/id_rsa.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any
that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it
is to install the new keys
root@10.149.245.208's password: <Password入力>
Number of key(s) added: 1
Now try logging into the machine, with:   "ssh '10.149.245.208'"
and check to make sure that only the key(s) you wanted were added.

# ssh root@10.149.245.208
Last login: Wed Dec 11 16:53:05 2019
```

SECTION 01

Dockerネットワークを使いこなそう

Dockerで構築されるアプリは複数のコンテナから構成されるのが一般的です。そのためには、Dockerの内部ネットワークを介したコンテナ間の接続などが必要になります。また、外の世界からコンテナにアクセスを行わせるためには、コンテナのポート公開などが必要です。Dockerらしいネットワークの使い方を理解することで、正しい設計を行えるようになります。

◎ Dockerネットワークの基礎を知ろう

2章で構築したWebサーバー（リバースプロキシー）とアプリサーバーのように、コンテナ間はネットワーク経由で通信ができます。また、Pythonコンテナ内でpipコマンドで外部からFlaskパッケージをインストールできたように、このネットワークはホスト外のネットワーク（自宅のLANなど）にも接続されています。

Dockerエンジンはコンテナの実行環境だけでなく、コンテナが利用するネットワークも提供しています。Dockerが持つネットワーク一覧は**「docker network ls」コマンド**で確認できます。下記のネットワークはすべてDockerがデフォルトで持つものであり、ユーザーがネットワークを作成した場合はそれも表示されます。

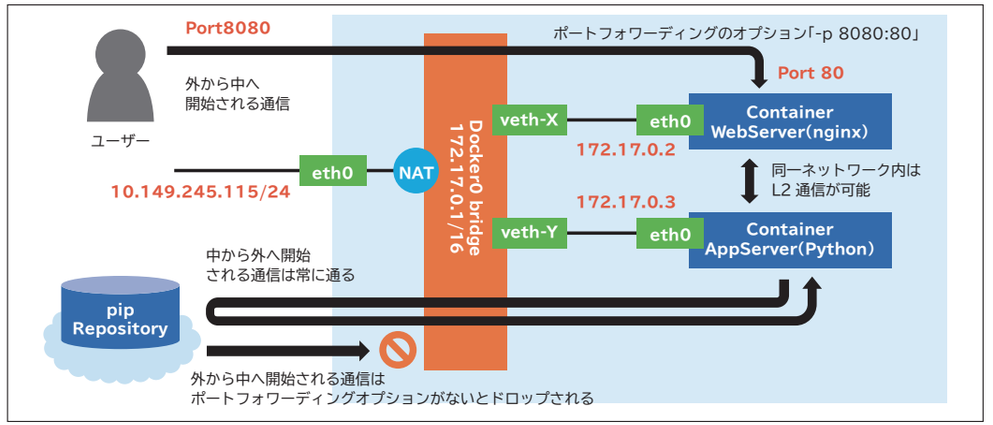
図3-1 ネットワーク一覧を表示

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
0aa42548fca8        bridge              bridge              local
408d3e2dd406        host                host                local
f510c2515b57        none                null                local
```

Dockerのネットワークの基本はNATです。特殊な構成であるDocker Desktopはもう少し複雑ですが、Dockerホストは自身が持つネットワークインターフェースをNATの外部インターフェースとして使い、NAT内のネットワークに内部IPアドレスを持たせたコンテナを接続しています。

上記の一覧にある「bridge」は、名前はスイッチのようですが、実体は**Dockerが提供するデフォルトのNAT用のネットワーク**です。このBridgeと2章で構築したWebサーバー（nginxのリバースプロキシー）とアプリサーバー（Python Flask）の構成は以下のようになっています。

図3-2 ブリッジとコンテナアクセス



Webサーバーからアプリサーバーへの通信は、この内部ネットワーク内の通信です。同一ネットワーク内の通信ですので、**コンテナ間は内部IP（正確にはARP解決とMACアドレスによるL2転送）で通信**を行います。そして、内部にあるコンテナから外への通信は、NATの**「IPマスカレード」機能**を使ってアドレス変換して行います。そして返りの通信はIPマスカレード機能で元の内部アドレスに変換されて、送信元のコンテナに送り届けられます。

たとえば、アプリサーバーはpipコマンドでインターネット上からモジュールをインストールしましたが、これは**デフォルトゲートウェイ（bridgeの内部インターフェース）**でアドレス変換されてホストのIPとして外に通信（pipのリクエスト）され、戻ってきた通信（モジュールのデータ）が元のコンテナの送信元IPに変換されてアプリサーバーに送られるという流れです。これはみなさんの自宅ネットワーク内の機器が、ルーターを経由してインターネット上のサイトに接続する流れとほとんど同じです。

外部からコンテナに開始される通信も一般的なNATとまったく同じ仕組みで動きます。NATに特別な設定がなければ、外部インターフェース宛に届けられた通信はどの内部IP宛かを判別できないので破棄されます。もしコンテナを外部に公開するサーバーとして利用したいのであれば、「外部インターフェースのポートX宛に届いた通信は、内部IPであるYのポートZに転送する」と設定を加えることではじめて実現できます。これはNATの**「ポートフォワーディング」**と呼ばれる機能で、今までrunコマンドで利用してきたオプション「-p 8080:80」は、ホストのポート8080宛の通信をコンテナ（のIP）の80番ポートにポートフォワードするという設定でした。

runコマンドを使う際に接続するネットワークを指定しなければ、bridgeが接続されるネットワークとして選択されます。それ以外のネットワークに接続したいのであればオプションで指定する必要があります。これは後ほどサンプル付きで説明します。

SECTION 03

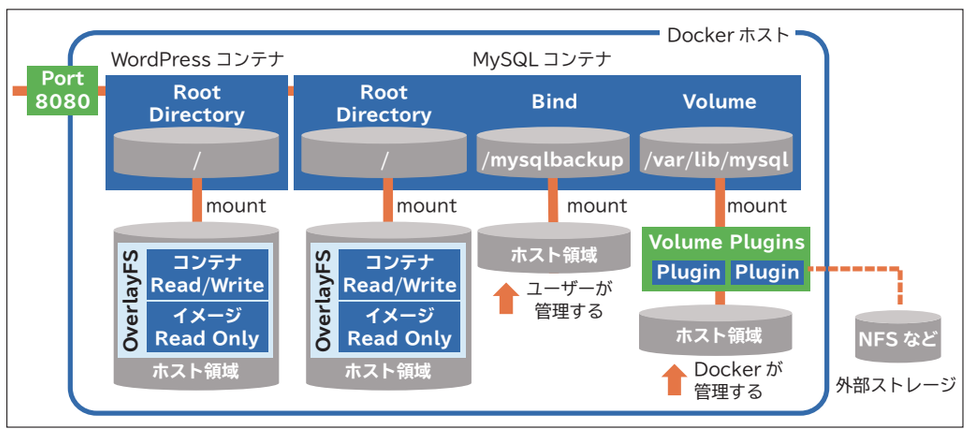
WordPressをデータ永続化してみよう

今まで学んできたネットワークとデータ永続化手法を使って、WordPress（ブログ）サイトの運用法を検討してみましょう。WordPressコンテナとそれが利用するMySQLコンテナ（データベース）を利用します。

◎ ここで作成するWordPressサイトの構造

今までの例ではコンテナを作っては壊しを繰り返してきましたが、実際にサービスを運用するのであれば「継続して運用しやすく、トラブルがあっても元に戻せる」ことが重要です。つまりデータ永続化はあたりまえで、それに加えてアプリのバージョンアップや、ソフトウェア／ハードウェア／オペミスに起因するトラブルに備えたデータベースのバックアップなども必要です。これらの操作運用を試すために、1つのホストで完結する下図のシンプルな構成を作成します。本来であればバックアップは同じホストではなく、別のホストやファイルストレージ／オブジェクトストレージなどに保存するのが妥当です。

図3-27 WordPressとMySQLのストレージ（コンテナ＋ボリューム）



まず前面のWordPressはアプリサーバーであり、たいした状態は持たないため永続化の対象とはしません。図にあるOverlayFSは先に話したイメージとコンテナのレイヤーを作るDockerのファイルシステムであり、ここは永続化されていません。ブログのデータを保存するMySQLは、データを失ってしまうと記事がすべて消滅するので、永続化が必須です。保存した領域をホスト上でユーザーが操作することはほとんどないため、バインドではなくボリュームを使います。

この永続化されるMySQLの領域は、問題なく運用できていれば使い続けることができます。ただし、ホストが壊れたり、コンテナ操作を間違えたり、間違って記事を消してしまった場合などは永続化された領域自体に問題が起きるため、記事は復旧できません。そのような状況でも復旧できるようにするために、**MySQLのデータをバックアップ領域に定期的にバックアップ**します。

構成をシンプルにするため、ここではWordPressの前面にWebサーバーは置きませんが、もし使うのであればnginxコンテナをリバースプロキシーとキャッシュサーバー（アプリサーバーの負荷を減らして高速化）として使うことなどが考えられます。

◎ データ永続化されたMySQLとWordPressを立ち上げよう

構築作業を開始しましょう。WordPressからMySQLへの接続に名前解決を使うので、新しくbridgeタイプのネットワーク「wp-net」を作成して、それにMySQLコンテナを接続します。

MySQLコンテナには、データベースのデータ領域（ボリュームmysqlvolume）を「/var/lib/mysql」としてマウントし、バックアップデータを置く領域（Bind）をコンテナの「/mysqlbackup」としてマウントします。そしてMySQLコンテナを使うのに必要な環境変数を4つ（MySQLのRootパスワード、MySQLパスワード、ユーザー名、データベース名）を与えます。MySQLのデータ領域のパスや必要とされる環境変数については、DockerHost上のドキュメントから確認したり、Dockerfileを読んだり、試しに起動して確認してください。MySQLに限らず有名な公式イメージであればWebで検索すれば使い方は見つかるはずです。なお、以下のサンプルにあるMySQLコンテナが使うバックアップ用のホスト側のディレクトリは、ご自身の環境のパスに置き換えください。

図3-28 ネットワークとMySQLのコンテナを作成

```
$ docker network create -d bridge wp-net
fbf12dd86edd0d225a18281e52686f77cbed43f00f38ebdd79e5f2591711cb59

$ docker container run -d --network wp-net --name mysql                    \
    --mount source=mysqlvolume,target=/var/lib/mysql                       \
    --mount type=bind,source=/Users/yuichi/mysqlbackup,target=/mysqlbackup \
    -e MYSQL_ROOT_PASSWORD=password -e MYSQL_DATABASE=wordpress             \
    -e MYSQL_USER=wordpress -e MYSQL_PASSWORD=password mysql:5.7.28
440e11e4c643ac24172d15e26de9bd32713d579fcc55a446cbdb670399a15a89
```

SECTION 01

Dockerfileの基本を知ろう

手動によるコンテナベースのアプリ開発／運用には手順書が欠かせませんが、手順書の不備や人的ミスにより継続的な開発／運用にトラブルが起きることも多いのです。何よりイメージの開発コストが大きくて負担となります。Dockerfileを使うことでイメージ作成を定義書通りに自動化できるため、これらの問題を解決できます。

◎ アプリ開発の流れとDockerfileの利点

Dockerベースのアプリを作成するのであれば、DockerHubで提供される既存イメージを使うだけでなく、自分でイメージを作成する必要があります。これは2章で紹介した「コンテナを作成して、docker image commitコマンドでイメージ化」という手法でも実現できますが、「Dockerfile」と呼ばれる機能がイメージ開発では一般的に使われます。

Dockerfileはイメージを作成するための手順書（設計図）であり、それを使った自動でビルド（イメージ作成）を実施する手法です。**「docker image build」コマンド**を使うと、Dockerfileに書かれた通りにDockerがイメージを自動で作成します。

1～3章の内容をきちんと把握できているのであれば、Dockerfileの文法さえ覚えてしまえばビルドは実施できるようになります。ただ、そもそもどのような状況でDockerfileを使うかを理解していないと、「何となく使ってみました」という域を超えません。勘と自己流で正しい正しい使い方にたどり着くには時間がかかりますので、2章のcommitを使った開発スタイルと比較しながらDockerfileの使い所を説明します。

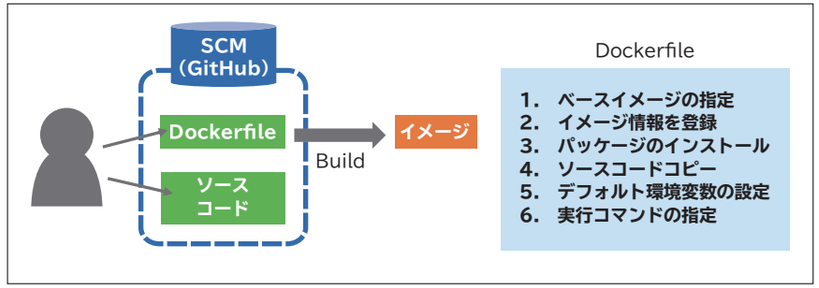
まず規模の大小に関わらず開発には流れがあり、おおまかにシンプル化すると以下のようなものになります。

1. 既存の開発環境（自分のPC上）を整える
2. 開発環境上でコードを書いて動作検証。不足があればステップ1に戻る
3. 成果物ができたら、リリースの準備に入る
4. 新規に本番環境をセットアップするか、既存の本番環境に変更を加える
5. 開発の成果物を本番環境に移行し動作させる
6. 機能更新やインフラのアップデートが必要となり、ステップ1に戻る

ここで着目してほしいのは、**開発環境は必ずしもきれいな状態とは限らず、構築手順も確立されていない**という点です。まったく新しいPCで構築手法が確立されたプロジェクトを開発すればきれいな環境となりますが、1～2年ほど他のプロジェクトの開発に利用したPCで別プロジェクトの更新版サービスの開発を開始することもよくあります。サービスを構築する手順がわかっていたとしても、そのような関係ないアプリが多数インストールされている汚い環境を更新する際に通用するかはわかりません。つまり、本番環境でサービスを更新する際にトラブルが起きる可能性が高いのです。

このような問題はDockerとDockerfileを使うことで解決できます。Dockerfileは先に説明したようにイメージを作成するための手順書ですので、そこに**書かれた通りにイメージが「新規」に作成されます**。日本語で書かれた手順書であれば曖昧な点があったり、オペミスによる構築失敗がありえますが、正しく書かれたDockerfileによるイメージ（アプリとその環境）の構築は必ず成功します。イメージの作成は常にクリーンな状態から行われ、それがコンテナとして展開される場合も新規の環境なので、変更などに気を配る必要はありません。

図4-1 Dockerfileを使ったイメージのビルド



似たような手法はDockerの登場前から実施されていますが、たとえば新規に展開された仮想マシンに対してシェルスクリプトで環境構築を行うのは時間と労力がかかります。アプリのコードに変更が発生したり、構築に失敗した場合にもう一度構築を実施するのは大変です。一方、DockerとDockerfileによるイメージ作成であれば、ビルドコマンドを使えば全自動でイメージ作成が始まりますし、ビルドに必要な時間は、後述するキャッシュ機能のおかげでスクリプトより圧倒的に短くなります。Dockerを導入したのであれば、開発にDockerfileを使わない理由はありません。

SECTION 01

Docker Composeを使ってみよう

Dockerfileを使ってイメージ作成手順を定義できたように、Docker Composeを使うことで複数のコンテナから構成されるアプリをどう展開するかを定義できます。大量のdockerコマンドを使って複雑な構成を作成することはトラブルの元ですので、シンプルな1コンテナのアプリの展開以外はComposeを使うことをおすすめします。

◎ Composeを使ってコンテナ利用法を定義できる

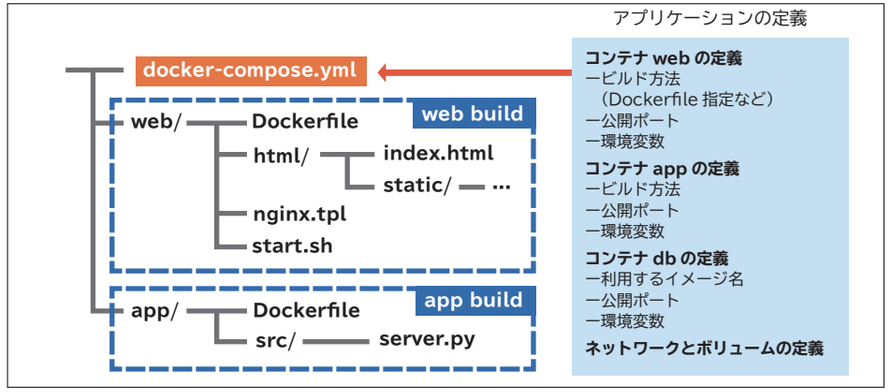
Dockerを使ったアプリは複数のコンテナを使って構成されることが一般的です。今までの「リバースプロキシとアプリサーバー」や「WordPressとMySQL」のように、コンテナAがコンテナBを使ったり、コンテナAとBが互いに利用し合うような構成です。このような構成にするために、今までは**別々に作成したイメージを1つずつ起動して、環境変数により連携させるという方式**でアプリを構築していました。

このような手動でのコンテナ間の連携方法は「どのように使うか」ということをドキュメント化しておき、複雑なオプションを正しくコマンド入力することでなりたっています。ドキュメントを書くのも人的コストが必要ですし、更新を繰り返すアプリでドキュメントが最新であるという保証もありません。長いrunコマンドで多数のコンテナを起動させたり、きちんとメンテされてない手順書にもとづく運用をしたりすることは、トラブルになりがちです。

Dockerfileを使うことでイメージをどのように作成するかを定義できましたが、同じように「イメージをどのように展開するか」ということを**Docker Compose（以下Compose）**という構成ファイルで定義できます。ビルドだけではなく複数のコンテナで構成されるアプリの設計やライフサイクル（ビルド／起動／停止）自体もDockerに管理させることで、素早く正しい展開をいつでも実施できるようになります。Dockerfileで単体イメージの開発コストを下げて、**Composeでアプリ全体としての開発コストと運用コストを大幅に下げられる**ということです。また、Composeの構成ファイルが正しければ、人的ミスによるコンテナの間違った運用も発生しにくいのでトラブル防止にもなります。きちんとDockerを開発／運用で利用するのであればComposeの利用はほぼ必須であり、多くのライトユーザーにとっては「Dockerホスト上でのComposeの利用」がコンテナの開発と運用スタイルにおけるゴールとなるかもしれません。

Composeがどのようなものか想像しづらいかもしれませんので、以下に4章で作成したnginxコンテナ（web）とflaskコンテナ（app）をComposeで構成する場合のディレクトリ構成図を示します。

図5-1 2階層アプリのComposeを使った構成



複数のコンテナ（イメージ）を束ねるアプリのディレクトリの下にComposeの構成ファイルがあり、そこでアプリをどのようにコンテナで構成するかを定義します。利用されるイメージがDockerfileでビルドされるのであれば、そのイメージごとに開発用ディレクトリを用意してCompose経由でビルドを実施することもできます。上記の図であればwebとappがサブディレクトリとして存在し、その中に4章のDockerfileでビルドした構成のファイル群があります。また、Composeではコンテナ群だけではなくそれらが使うネットワークやボリュームなども定義でき、アプリを構成するデータベースなどもコンテナで実現できます。こういったアプリの土台をコードベースで展開できるという点で、Composeは「**Infrastructure as a Code**」を実現するツールであるともいえます。

なお、このような複数の要素（コンテナ）をまとめて管理することを「**オーケストレーション**」と呼びます。Kubernetesはコンテナのオーケストレーションツールの代表格ですが、ComposeはKubernetesの10分の1程度の複雑さで、「1ホスト内限定」で使えるオーケストレーションツールと思えばわかりやすいかもしれません。

筆者は、小規模なDockerベースのアプリはもっぱら仮想マシン上にAnsibleでDockerホストを構築し、そこにComposeを使って展開しています。ストレージなどのデータもホスト上に保存しています。この構成をとると、ホスト障害（物理的に壊れたなど）のときにアプリが落ちるという弱点を「仮想化のHA（High Availability。別の物理ホストでマシンを再起動すること）」でカバーすることができ、データロスのリスクをDockerホストの状態をストレージスナップショット（仮想マシンのディスク状態を残すこと）で定期的に保存することで低減させています。KubernetesはComposeに比べて開発と管理の手間が増えるので、上記シナリオ以上の可用性やスケール性が必要になった場合のみ利用しています。

SECTION 04

Composeで本格的な3階層アプリを開発してみよう

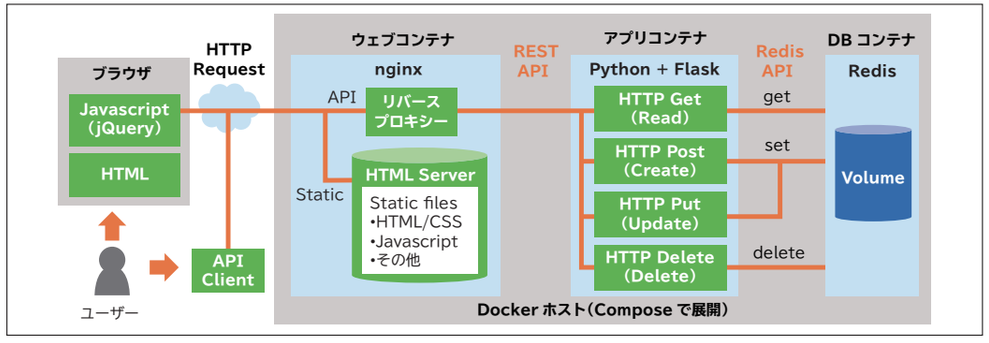
今まで学んできた内容の総復習としてWeb3階層構成（web、app、db）のアプリをComposeで作成します。新しい技術としてRedis（KVS）を使いますので、利用法の紹介をしたあとで、コードを提示しつつ開発の流れを解説します。このサンプルアプリは6章のDockerを使ったCI/CDでも利用します。

◎ 開発するアプリの構成について

本節ではWeb3階層のアプリをComposeで構築します。今までのサンプルに比べるとサイズが大きく複雑ですが、今後の章で学ぶDockerやKubernetesを使ったDevOps（CI/CD）で使う実用的な構成となります。プログラムの内容を厳密に理解しなくても試せるようにしていますが、どういうようなコードを書くか興味がある方もいるでしょうから中心的なコードを記載します。

構築するアプリは「**KVS（Key Value Store）**」というWeb開発でよく使われる技術を、**REST API（HTTPベースのAPI）**で提供するというものです。KVSもREST APIもDockerやWeb系の開発ではよく利用される技術なので、知らない人はこの機会に概要を把握されることをおすすめします。両者とも有名な技術なので、不明な点があればネットで簡単に調べられます。

図5-17 構築する3階層アプリの構成



このアプリではプログラムが利用することを想定したAPIサーバーの作成に加えて、それを人間が操作するためのGUI（ブラウザ画面）をHTMLとJavaScriptで構築します。

一番後ろにあるDBサーバーである**Redis**は公式イメージのRedisを使って展開します。ボリューム機能を使ってデータ永続化している以外は特に言及するべきところはありません。Redisの前面にいるアプリサーバーは、REST APIで受けた命令に従って、Redisを操作して結果を返すプロトコル変換のプロキシーとしての役割を果たしています。

フロントにいるnginxは静的なファイルへのアクセスはHTMLサーバー機能で処理し、APIへのアクセスはリバースプロキシーでアプリサーバーに転送する処理をします。静的ファイルにはAPIを呼び出すためのHTMLとJavaScriptのGUIの画面も含まれています。ユーザーはブラウザ画面を通してKVSの機能を利用することもできますし、自分のプログラムでREST APIを発行することでKVSの機能を利用することもできます。図の構成を作成するComposeファイルは以下の通りです。

リスト5-8 /chap5/c5kvs/docker-compose.yml

```
version: '3.7'
services:
  web:
    build:
      context: ./web
      dockerfile: Dockerfile
    depends_on:
      - app
    ports:
      - 8080:80
    environment:
      APP_SERVER: http://app:80
  app:
    build:
      context: ./app
      dockerfile: Dockerfile
    depends_on:
      - db
    environment:
      REDIS_HOST: db
      REDIS_PORT: 6379
      REDIS_DB: 0
  db:
    image: redis:5.0.6-alpine3.10
    volumes:
      - c5kvs_redis_volume:/data
volumes:
  c5kvs_redis_volume:
    driver: local
```

SECTION 01

継続的開発とデプロイについて知ろう

多くのアプリは開発して終わりではなく、最初のバージョンを公開したあとも新機能追加や不具合修正などが必要です。それらの変更を適用したソフトウェアを作成して公開するのは手間がかかりますので、CI/CDと呼ばれる自動化手法が広まってきています。本章ではComposeを使ったアプリでCI/CDを行う手法を紹介します。

◎ SCMとパイプラインを使ったCI/CDの実現

使い捨てではないアプリは、継続的に開発／デプロイ(公開)される必要があります。みなさんが利用されているスマホやPCのアプリも更新が頻繁に発生しますし、利用しているWebサービスの画面や機能も更新され続けています。これは裏側でアプリが開発され続けており、新しいバージョンのアプリが本番環境にデプロイされているためです。

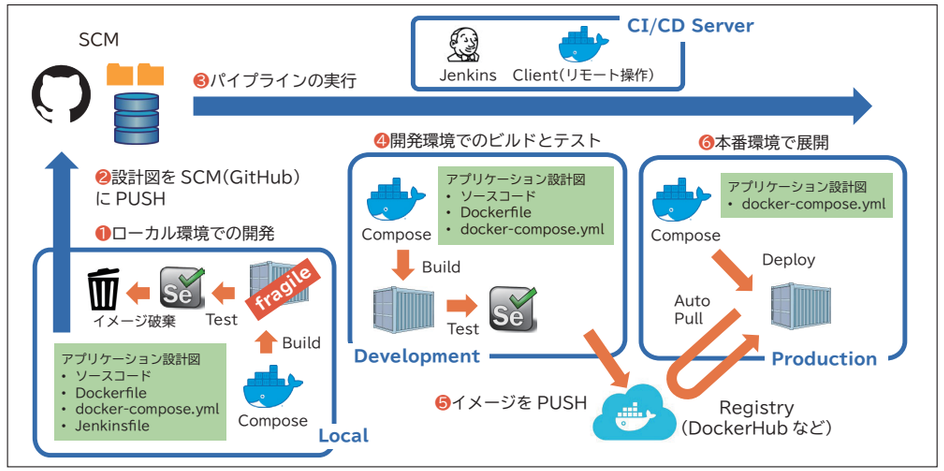
アプリをDockerベースで構築する場合も同じで、Dockerのイメージを継続的に開発し、本番環境で古いコンテナを新しいイメージのコンテナに置き換えないとアプリを更新できません。本章では前章で学んだComposeと「**CI/CD（Continuous-Integration/Continuous-Deployment）**」と呼ばれる手法を使うことで、Dockerベースのアプリ（5章のKVS）を効率的に開発／デプロイする手法について学びます。

CI/CDを構築するツールはさまざまですが、最も重要なものを挙げろといわれれば、4章で学んだ**Git（GitHub）に代表されるSCM（Source Code Management）**と、本章で学ぶ**Jenkinsに代表される「パイプライン」**です。SCMを適切に使えば過去から現在までのソースコードを管理しつつ、複数人で開発作業を行うことも容易となります。そしてパイプラインを導入することによりアプリのビルドからデプロイ（もしくはその一歩手前）までを自動化できるので、変更を加えたアプリのコードをリリースする作業が容易となります。

本書で学んでいるDockerやKubernetesは、SCMともパイプラインとも非常に相性がよいツールです。アプリのソースコードに加えてイメージをビルドする手順もDockerfileとしてテキストで定義でき、イメージ群をどのように使うかということをComposeファイルでテキストして定義できます。ソースコードもDockerfileもComposeファイルもSCMでバージョン管理と共有ができますし、パイプラインによるビルドとデプロイもComposeを使うことで複雑さを大幅に減らすことができます。

以下に本章で構築するCI/CD環境の全体像を記載します。

図6-1 本章で構築するDockerを使ったCI/CD構成



図の左下がみなさんのPCにあたるローカル開発環境です。複数の開発者がいる場合はそれぞれがこの環境を持っています。アプリの開発は、このローカル開発環境で行われます。

この開発環境で作成されたコードはSCM（GitHub）にPushされます。SCMを使うことによって複数の開発者が分担作業を行うことができるようになり、**別々に開発されたコードがマージされて1カ所に管理**されます。また変更なども記録されるので、コードが壊れてしまった場合も原因調査や復旧をしやすくなります。

SCMにコードがPushされると図の右側のCI/CD Server（Jenkins）上のパイプラインが起動され、自動でビルドからデプロイまでの作業が実施されます。詳細は後ほど触れますが、パイプラインで何をするかは**パイプラインの定義ファイル**に書くことが一般的です。Jenkinsであれば「Jenkinsfile」と呼ばれる定義書に書きます。パイプラインが起動されると、定義ファイルが読み込まれて記載される作業を順番に処理していきます。Dockerを使ったアプリ開発であれば、図にあるように**Jenkinsがリモートの Dockerホストを Composeコマンドで操作することでビルドを実施**します。イメージ作成に成功したらコンテナのテストを実施し、テストにパスしたら本番に使えるイメージということで、レジストリ（DockerHub）に作成したイメージをPushします。

次のJenkinsの仕事は、**本番環境に先ほど作成したイメージをデプロイする**ことです。Jenkinsが本番環境のDockerホストに対して、Composeコマンドで最新イメージを指定してデプロイ指示します。そうするとDockerホストが自動でイメージの取得をレジストリから行い、現在動いているコンテナを最新版のものと置き換えます。これでCI/CDのすべてのステップが完了します。

SECTION 01

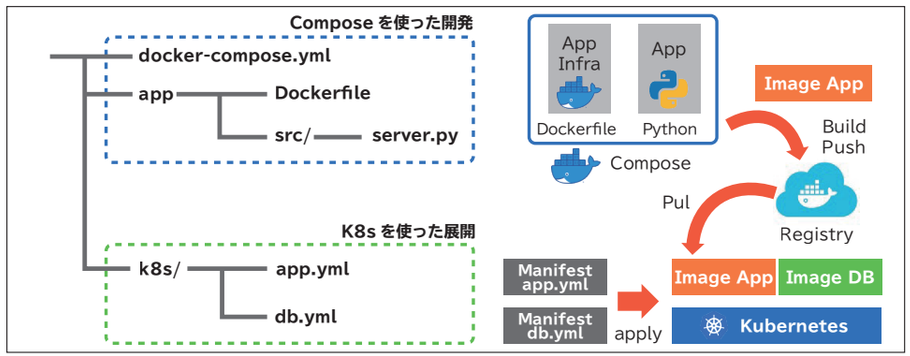
Kubernetes用イメージを開発しよう

K8sで独自のアプリを運用するのであれば、K8s用のDockerイメージの開発が必要となります。ここではDocker編で学んだ開発テクニックを踏み台として、Composeでイメージを開発してK8sで利用する流れを紹介します。Dockerで簡単な構成でイメージ開発とテストを行い、そのあとでK8sの複雑な構成で利用すると開発コストが少なくなります。

◎ 開発の流れ

前章のようにnginxなどの既存の公式イメージを使うのであればイメージの開発は不要です。ただ、独自サービスをK8s上のコンテナで提供したいのであれば、利用するイメージ群の開発が必要です。ここではその題材としてユーザーからのアクセス数をカウントするアプリ（Flaskポッドと Redisポッド）を以下の図の流れで作成します。

図8-1 Composeによるイメージ作成とK8s上へのコンテナ展開



開発作業に取り掛かる前に図の左側にあるディレクトリの構成を確認します。ディレクトリ内はおおまかに**Compose用のファイル／ディレクトリ**と**K8s用のファイル／ディレクトリ**に分けられています。前者がアプリのルートディレクトリにあるComposeファイル（docker-compose.yml）と、アプリを構成する各イメージのディレクトリ（ここではapp）です。そして後者がK8sのマニフェストファイル群を持つk8sというディレクトリになります。5章で学んだComposeの開発用ディレクトリ構成に、7章で学んだK8sのマニフェスト用のディレクトリを追加しただけです。

K8s用のイメージと運用を行うためにどういったディレクトリ構造を作るかは組織次第です。ただ、どのようなディレクトリ構成をとろうとDevOpsスタイルで開発と運用を行うのであれば、**アプリの開発用コード（ソースコードやDockerfileなど）と運用のコード（マニフェスト）を同一のディレクトリに入れる**とSCM（GitHub）で管理しやすくなります。

図の右側が開発からデプロイまでの流れとなります。Composeを使ってイメージをビルドしたあとで、イメージをレジストリ（DockerHub）にPushします。レジストリにPushされたら7章のnginxやWordPressと同じように自動Pullでappとdbポッドを展開できます。この流れ自体は単純なのですが、注意が必要なのは自作イメージをK8sに展開するのはDockerに比べると手間がかかるということです。Dockerはローカル環境でビルドしたものをすぐにコンテナ化できるのに対し、K8sは**作成したイメージをDockerでレジストリにPushしてK8sがそれをPullして利用する**という流れを避けることができません。開発したソフトウェアが修正なしに動いてテストにパスする可能性は低いので、おそらく何度もイメージのビルドと実行を繰り返すことになります。そのため、いきなりK8sでテストと実行をするのではなく、**簡易構成をComposeで動作確認しながら作成し、それが動いてからK8sに持っていく**という開発手法が、展開コストが低くおすすめです。第6章で学んだCI/CDをうまく使ってください。

◎ 作成するアプリの仕様

アプリのコードを提示します。ブラウザ上で背景色をランダムに生成して「Version:1, AccessCount:X, HostName:Y」というメッセージを表示するFlaskのアプリサーバーです。メッセージのXは今までアクセスされた回数で、Yはホスト名（コンテナ名）です。ページにアクセスされるたびに、Pythonが裏側にいるRedis（DBサーバー）にアクセス数の問い合わせを行います。そしてRedisに+1したアクセス数を再登録してから、クライアントにページ情報を送ります。なお、紙面の都合で接続先のDBホスト名以外のパラメーター（ポート番号など）はコードに埋め込んでいます。

リスト8-1 /chap8/c8dev/app/src/server.py

```
import flask, os, re, redis
from random import randint
REDIS = redis.Redis(host=os.environ['REDIS_HOST'], port=6379, db=0)

COLOR = f'rgb({randint(0, 255)},{randint(0, 255)},{randint(0, 255)})'
HTML = '''<!DOCTYPE html>
<html><body style="background-color:{}">
  <h1>Version:1, AccessCount:{}, HostName:{}</h1>
</body><html>'''
```

SECTION

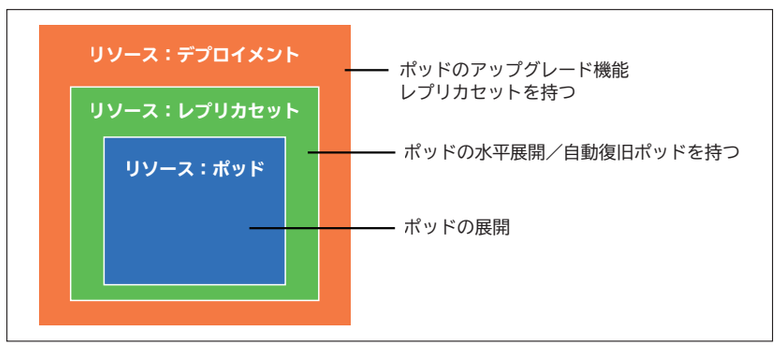
02 デプロイメントを使ってみよう

ポッドを柔軟に利用するためにはデプロイメントリソースを使います。水平に同一機能を持つ複数のポッドを展開したり、アプリのバージョンアップを柔軟に実現できます。実運用ではポッドではなくデプロイメントを利用するのが一般的です。本節ではこれに加えてブルー／グリーンデプロイメントなどについても扱います。

◎ デプロイメントリソースの概要

ポッドとサービスを使ってアプリをK8s上で動かすことはできますが、実運用場面ではポッドを機能拡張したデプロイメントというリソースを使うことが一般的です。デプロイメントは内部的にレプリカセットというリソースを持っており、それがポッドリソースを持つという階層構造となっています。以下にこの3つのリソースの関係を図にまとめます。

図8-5 デプロイメント、レプリカセット、ポッドの関係



デプロイメントは内部に下位リソースとして「レプリカセット」というリソースを持ち、それが水平スケール（複数台のポッドを展開して負荷分散）や自動復旧機能（壊れたポッドを新品のポッドに交換する）を担当しています。レプリカセットを持つデプロイメントの主要機能は、ポッドのアップグレード（ソフトウェアのバージョンアップなどが発生した際に、古いポッドを新しいポッドに賢く入れ替える）です。マニフェストファイル内でのレプリカセットの設定はデプロイメントと同じ階層で行うため意識しないかもしれませんが、ポッドの設定はデプロイメントの下に「template」という形で行います。今までデプロイメントではなくポッドリソースを使ってきたのは、そちらのほうがシンプルでわかりやすいという理由からです。

以下に先ほど開発したカウンターアプリをデプロイメントで展開するマニフェストを記載します。specからtemplateまでの間がデプロイメントの新しい設定です。デプロイメントのtemplate以下の設定が前節のPodとまったく同じで、後半のサービスも前節とまったく同じです。

リスト8-6 /chap8/c8deployment/k8s/app.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app
spec:
  replicas: 5
  strategy:
    rollingUpdate:
      maxSurge: 50%
      maxUnavailable: 0%
  minReadySeconds: 5
  selector:
    matchLabels:
      pod: app
  template:
    metadata:
      name: app
      labels:
        pod: app
    spec:
      containers:
      - name: app
        image: yuichi110/c8dep_app:v1.0
        ports:
        - name: http
          containerPort: 80
        env:
        - name: REDIS_HOST
          value: db

---
apiVersion: v1
kind: Service
metadata:
```