1.1 Kubernetesとは

1.1.1 コンテナオーケストレーションとKubernetes

Kubernetes（クーバネティス）（**図1-1**）は、コンテナ化されたアプリケーションのデプロイ、設定、管理を自動的に行うオープンソースソフトウェア（OSS）です。このようなソフトウェアを「コンテナオーケストレーションツール」と呼んでいます。

図1-1 Kubernetesのロゴマーク[注1]

開発者は、コンテナ化されたアプリケーションから構成されるサービスのあるべき状態を設定ファイルに宣言的に記述し、複数のノード（VMや物理マシン）から成るKubernetesクラスタに反映します。そして、Kubernetesは反映された設定ファイルに従ってクラスタ上にアプリケーションを展開します。Kubernetesの抽象化機能、自己回復機能などによりスケーラブルで信頼性の高いサービスをクラスタ上に構築できます。また、Webサービスのようなステートレスなアプリケーションから、データベースのようなステートフルなアプリケーションまでをKubernetesで構築できます。

そのほかにも、マイクロサービス指向なサービスを開発するためのサービスディスカバリや、クラスタを論理的な複数のクラスタに分割する名前空間、柔軟な認証・認可など、サービスの成長はもちろん、開発チームの拡大にも対応できる機能を持っています。小さなサービスからとても大きなサービスまで恩恵を受けることができ、あらゆる規模のサービスの構築に適しています。

Kubernetesは、Googleの社内システムであるBorgにインスパイアされ、当初はGoogleが開発していましたが、その後Linux Foundation傘下の組織であるCloud Native Computing Foundation（CNCF）に寄贈され、今では多くの企業が参加するコミュニティ主体のプロジェクトとなっています。10年以上に渡ってコンテナを本番環境で運用してきたGoogleの経験と、コミュニティの優れたアイデアと手法がKubernetesに組み込まれています。

注1 https://github.com/kubernetes/kubernetes

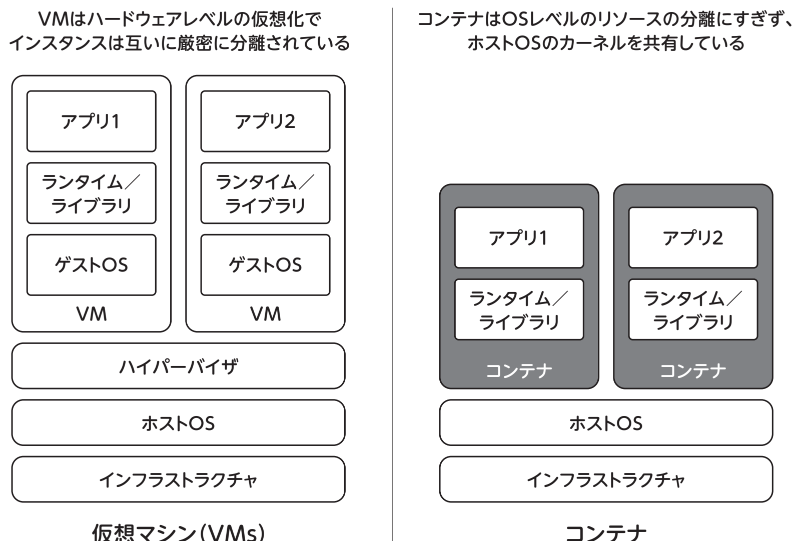
1.1.2　Linuxコンテナ技術とDocker

Kubernetesは、複数のアプリケーションを複数のマシンから構成されるクラスタにデプロイし、管理するソフトウェアです。開発者はアプリケーションのデプロイを指示するだけで、あとはKubernetesが適切なマシン上でアプリケーションが実行されるようにスケジュールします。その結果、個々のマシン上ではさまざまな種類のアプリケーションが複数実行されることになります。

通常、アプリケーションの実行は言語ランタイムやライブラリ、実行環境の共有リソースであるネットワークポートなどに依存します。たとえば、あるアプリケーションがNode.js v8.11に依存していたとします。もし同じマシン上で実行されるようにスケジュールされた別のアプリケーションが、Node.js v10.9に依存していたとしたらどうでしょうか。また、共有ライブラリやコマンドラインツールも一般的にマシンの共有コンポーネントであり、インストールされているバージョンで正しく実行できるかはわかりません。このようなことから、それぞれのアプリケーションの実行環境を分離しなければいけません。Kubernetesは、このアプリケーション実行環境の分離にコンテナ技術を利用しています。

これまで実行環境の分離と言えば、VM（Virtual Machine、仮想マシン）が利用されてきました。これまでのVMを利用した環境では、アプリケーションごとにVMを用意するなどしてアプリケーションの実行環境を分離していました（**図1-2左**）。しかし、当初は小さかったサービスも、成長とともに管理するVMが増加します。フロントエンドやバックエンド、データベースやキャッシュサーバなどの役割が異なるVMは、それぞれに特化した設定が必要な場合が多く、構築やメンテナンスにかかる人的リソースが増加していきます。また、それぞれのVMでの余分なハードウェアリソースも無視できないものになってきます。

図1-2　仮想マシンとコンテナの比較



そこでVMに代わり、アプリケーションの実行環境の分離にLinuxコンテナ技術を利用します。コンテナ技術を利用することで、1つのVM上で複数の異なる要件を持つアプリケーションの実行環境を分離して実行できます（**図1-2右**）。この際、VMのホストOSからみると、アプリケーションのプロセスはほかのプロセスと同様に実行されているように見えます。しかし、アプリケーションのプロセスからみると、ホストOS上のプロセスはもちろん、ほかのコンテナのプロセスも確認することはできません。そのほか、ファイルシステムやユーザ、ネットワークポートも分離されていることから、専用のVM上でアプリケーションを実行するのと同様の使い勝手です。このことからコンテナ仮想化と呼ぶこともあります。

VMとコンテナを比較すると、VMは、CPUやメモリ、ストレージなどのハードウェアリソースを仮想化していることから、各インスタンスが独立したカーネルを持ち互いに厳密に分離されています。そのため、インスタンスの起動には、完全なOSの起動が必要となり実行が遅いです。一方、コンテナはプロセスごとのリソースの分離と制限にすぎません。そのため、同一ホスト上のコンテナは、ホストOSのカーネルを共有しており厳密に分離されていないことから、VMと比較してセキュリティに弱みがあります。しかし、コンテナの起動はOSの起動が不要なことから非常に高速であり、通常のプロセスと同等の速さで実行されます。

2018年現在では、LinuxコンテナでVMレベルの分離を実現するためのソフトウェアが登場しています。その1つが、Googleが開発するgVisor[注2]です。gVisorはユーザランドで実行されるカーネルで、ホストカーネルへのシステムコードを先に受け取り、その後ホストカーネルに渡すことでサンドボックス化を実現しています。gVisorは、Google App Engineがサポートするいくつかの言語の実行環境として採用されています。

Linuxコンテナを実現する機能

Linuxコンテナの実現には、Linuxカーネルの2つの機能が利用されています。

1つめは、Linux Namespacesです。これは、プロセスに対して専用の名前空間を作成する機能です。プログラミング言語における名前空間は、関数や変数の名前が衝突しないように分離しますが、ここではファイルやプロセス、ネットワークなどのLinuxのリソースを分離します。これによってアプリケーションの分離を実現しています。

2つめは、Linux Control Groups（cgroups）です。これは、プロセスに対して利用できるリソース（CPUやメモリなど）を制限します。この制限によって、同一のマシン上に複数のアプリケーションを実行した際に一部のアプリケーションがリソースを使いすぎる（うるさい隣人）などの問題を防ぎます。

これらの機能によって、OSレベルでアプリケーションの分離を実現しています。

注2　https://github.com/google/gvisor

Dockerとは

Linuxコンテナ技術はこれまでも利用されていましたが、Docker（**図1-3**）の登場で広く利用されるようになりました。

図1-3　Dockerのロゴ注3

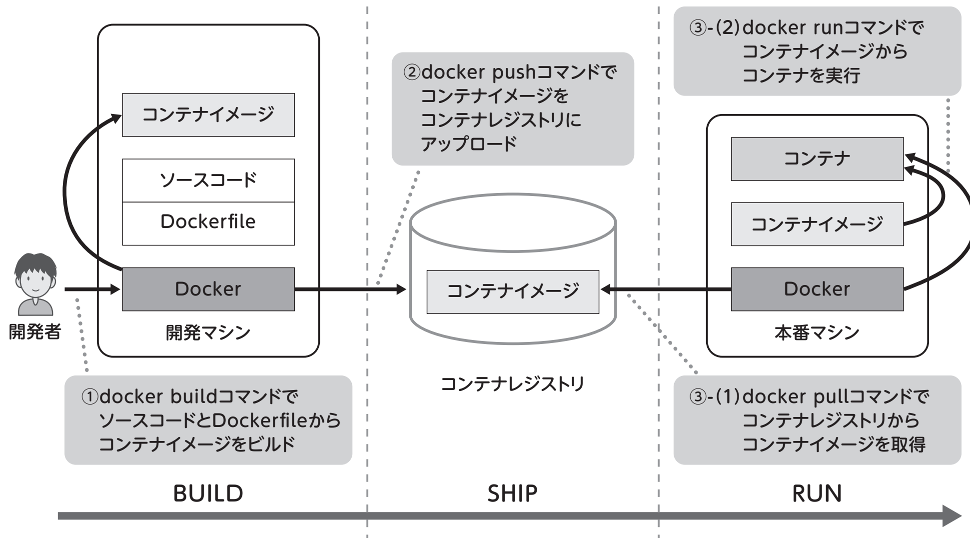
Dockerは、Linuxコンテナ技術とレイヤ（層）のファイルシステムを利用して、アプリケーションのパッケージングと配布、実行を行うソフトウェアです。アプリケーションの実行に必要な言語ランタイムやライブラリ、ソフトウェアなどをすべて含んだコンテナイメージという1つのバイナリのような塊（かたまり）として、アプリケーションを管理、実行できるようにしたことで、いつどこで実行しても必ず同じものとなるイミュータブル（不変）でポータブルな性質をアプリケーションにもたらしました。コンテナイメージにアプリケーションが依存するすべてが含まれていることから、ローカル環境と本番環境でまったく同一のアプリケーションを実行できます。これまでよく遭遇した開発環境では正常に実行できたが、本番環境ではうまく実行されなかったなどの環境の違いによる問題は起きにくくなります。

Dockerの本質は、開発したコード、アプリケーションの本番環境への高速なデリバリにあります（**図1-4**）。コンテナイメージの作成から本番環境で実行するまでの流れは次のとおりです。

①BUILD：アプリケーションの実行に必要なすべてを含むコンテナイメージのビルド
②SHIP：コンテナイメージを配布するためのレジストリへのアップロード
③RUN：本番環境でのコンテナイメージの取得と実行

注3　https://www.docker.com/

図1-4 コンテナイメージのビルドから本番マシンで実行されるまで



BUILDフェーズでは、Dockerfileと呼ばれるファイルをもとにコンテナイメージを作成します。Dockerfileは、UbuntuなどのOSやNode.jsなどの特定の言語ランタイム向けの基本となるコンテナイメージ（ベースイメージと呼びます）をベースにして、ファイルを追加するCOPYやコマンドを実行するRUNなどの複数の命令をステップとして記述します。**リスト1-1**は、Node.jsのアプリケーションの例です。

リスト1-1 Node.jsアプリケーションのDockerfile

```
FROM k8spracticalguide/node:10.11.0

WORKDIR /src/app

COPY package*.json ./

RUN set -x && \
    npm install --production

COPY . .

ENV NODE_ENV production
USER node
EXPOSE 8080
CMD ["npm", "start"]
```

このテキストファイルをDockerfileというファイル名で保存します。その後、Dockerのコマンドラインインターフェース（CLI）であるdocker buildコマンドを利用してビルドすることでコンテナイメージを作成します。

```
カレントディレクトリのDockerfileを利用してコンテナイメージを作成する
-t（--tag）オプションでイメージ名を指定する
$ docker build -t registry.example.com/demo/app:v1 .
```

コンテナイメージの作成は、ラップトップなどのローカルの開発マシンでも行えますが、テスト用途以外の本番マシンで実行するためのコンテナイメージは、一般的にGitなどのソースコードリポジトリへのプッシュを契機に継続的インテグレーション（CI）システム上でビルドを実行します。これは、コンテナイメージの作成を一般的なソフトウェア開発方法と同様に、継続的にテストできることを意味します。

次にSHIPフェーズです。ここではビルドしたコンテナイメージを、docker pushコマンドを利用してコンテナレジストリにアップロードします。コンテナレジストリはコンテナイメージの保存、管理、配信を行うシステムです。

```
ローカルのコンテナイメージをリモートのレジストリにアップロード
$ docker push registry.example.com/demo/app:v1

一度アップロードすればどこからでも取得できる
$ docker pull registry.example.com/demo/app:v1
```

これにより、ほかのホストからアップロードしたコンテナイメージを取得できるようになります。また、コンテナイメージは環境ごとに用意する必要はなく、同じコンテナイメージを開発や本番などのすべての環境で利用できます。コンテナレジストリは、パブリックなSaaSであるDocker Hub[注4]を利用したり、自身で構築することでプライベートなレジストリを用意したりすることもできます。レジストリには認証機能が付属していることが多く、docker loginコマンドで事前に認証を行います。

最後にRUNフェーズです。ここではdocker runコマンドを利用します。コンテナレジストリからコンテナイメージを取得して実行します。コンテナはLinuxコンテナ技術を利用してホスト環境やほかのコンテナから分離されています。これにより、いつどこで実行しても同じものが実行されるようになっています。

注4 https://hub.docker.com/