初心者でもわかる Kubernetesとコンテナ基礎

The Linux Foundation　水澤泰敬

THE LINUX FOUNDATION

自己紹介

水澤泰敬（みずさわやすたか）

日経新聞関連企業などへの転職を重ねたのち、フリーランスを経て起業。情報まとめサイト運営に携わる。キュレーションサイトへの転職、事業譲渡によるTSUTAYA関連企業への転籍を経て再びフリーランスとなる。

Linux関連資格取得を機に、日本のエンジニア育成に携わりたいという思いから、エンジニアスクール講師として活動ながらLinux技術者認定資格取得を目指す人を応援する情報サイト「リナスク」を運営。

クラウドネイティブ時代にコンテナオーケストレーションの知識が必須と判断し、独学でKubernetes関連資格であるCKAとKCNAを取得。

趣味はモバイルガジェットいじり、音楽鑑賞、楽器、テレビゲーム。

所有資格：

・CKA（認定Kubernetes管理者）

・KCNA（Kubernetesクラウドネイティブアソシエイト）

・LinuC／LPIC

・情報セキュリティマネジメント

・個人情報保護士　等

THE LINUX FOUNDATION

アジェンダ

1. クラウドネイティブとは
2. コンテナとコンテナオーケストレーション
3. Kubernetesとは
4. Kubernetesの学習方法と関連資格の紹介

クラウドネイティブとは

Linux Foundationとは

- 開発者がオープン テクノロジー プロジェクトでコードを開発、管理、スケールさせるための中立的で信頼できるハブを提供。
- オープンソース ソフトウェア、オープン スタンダード、オープン データ、オープン ハードウェアに関するコラボレーション拠点。
- ベストプラクティスを活用し、コントリビュータ、ユーザー、ソリューション プロバイダーのニーズに対応し、オープンなコラボレーションのための持続可能なモデルを構築。
- オープンテクノロジーと商業的な採用を加速させる、Linuxをはじめとする最も重要なオープンソースプロジェクトを900以上ホスト 。
- 傘下の一つにオープンソース団体「Cloud Native Computing Foundation (CNCF) 」。

THE LINUX FOUNDATION

CNCFとは

- クラウドネィティブ コンピューティング環境の普及を目指す団体

  ↓

  クラウドを前提に、設計／構築／開発／運用するコンピューティングスタイル　※参考：マイクロサービスの仕組み（翔泳社刊）

- オープンソースでベンダー中立プロジェクトのエコシステムを育成・維持して、クラウドネィティブパラダイムの採用を促進。最先端のパターンを民主化し、これらのイノベーションを誰もが利用できるようにする

- Kubernetes を含む、多くのプロジェクトのベンダ中立なホーム

クラウドネイティブ(アーキテクチャ)の基本

- パブリッククラウド、プライベートクラウド、ハイブリッドクラウドなどの近代的でダイナミックな環境において、スケーラブルなアプリケーションを構築および実行するための能力を組織にもたらす技術。

- コンテナ、サービスメッシュ、マイクロサービス、イミュータブルインフラストラクチャ、宣言型APIなどの手法により、回復性、管理力、および可観測性のある疎結合システムが実現。堅牢な自動化と組み合わせることで、インパクトのある変更を最小限の労力で頻繁かつ予測どおりに行うことができる。

CNCF Cloud Native Definition v1.0
https://github.com/cncf/toc/blob/main/DEFINITION.md

クラウドネイティブアーキテクチャの特徴

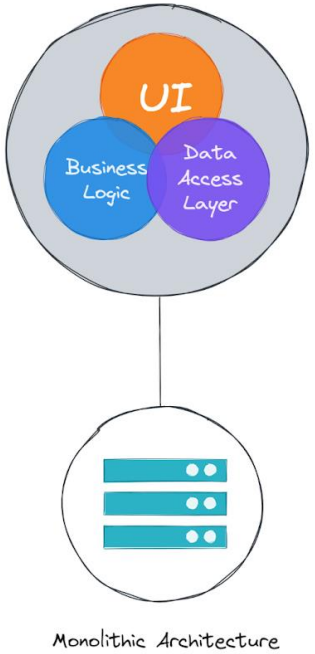
- ✓ ハイレベルな自動化 High level of automation
- ✓ 自己修復 Self healing
- ✓ 拡張性 Scalable
- ✓ 費用効率 (Cost-) Efficient
- ✓ メンテナンス性 Easy to maintain
- ✓ デフォルトの安全性 Secure by default

クラウドネイティブ理解のために重要な要素

- モノリシックアーキテクチャ
- マイクロサービスアーキテクチャ
- オートスケーリング
- サーバーレス
- オープン スタンダード

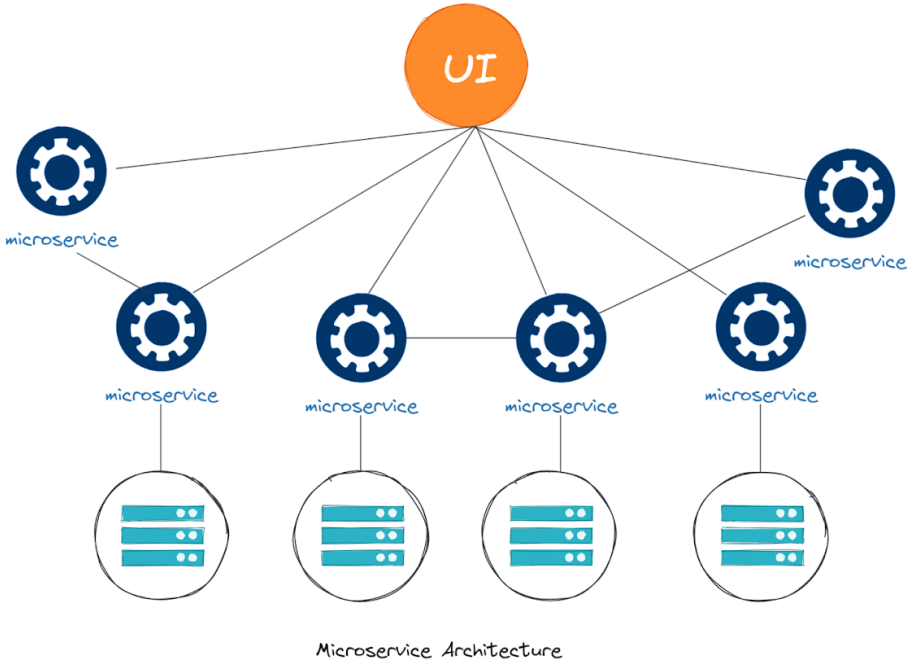
モノリシックアーキテクチャ

- アプリケーションに、タスクを満たすために必要なすべての機能とコンポーネントが含まれている
- シングル コードベースで、サーバー上で実行できるシングル バイナリファイルとして提供
- システムのすべての機能をまとめてデプロイ



マイクロサービスアーキテクチャ

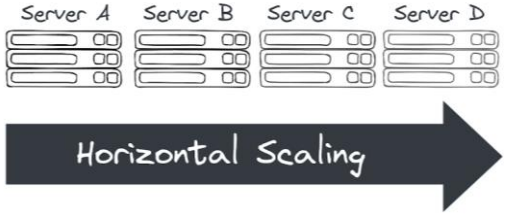
- 管理しやすくなるように小さいピースに分解
- ネットワーク上でお互いにやり取りが可能
- 機能ごとにグルーピングやスケールが可能



オートスケーリング

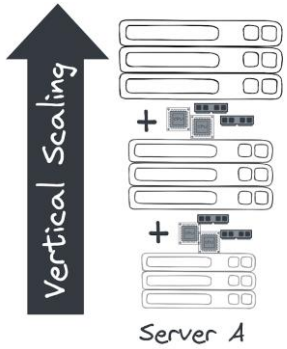
- **水平スケール/horizontal scaling**

  物理マシンや仮想マシン、アプリケーション プロセスなど、新しいコンピュータ リソースを追加生成



- **垂直スケール/Vertical scaling**

  基礎となるハードウェア サイズを変更
  基となるハードウェアスペックによって上限が決まる



サーバーレス

- 自社に物理的なサーバーを置かずに、クラウドサービスとしてサーバーを利用すること
- インフラは抽象化されており、開発者はzipファイルやコンテナイメージなどのコードをアップロードすることで、ソフトウェアをデプロイできる
- FaaS（Function as a Service）のような、機能の実行環境を提供するサービスが代表的

オープン スタンダード

- クラウドネイティブ技術は、オープンソースソフトウェアに強く依存しているため導入が簡単
- ベンダーロックインを防ぐ
- 誰でもプロジェクトに参加でき、さまざまな製品への実装も容易

（参考）

- OCI Spec: コンテナの実行、ビルド、配布方法におけるイメージ、ランタイム、配布に関する仕様
- Container Network Interface (CNI): コンテナのネットワーキング構築方法に関する仕様
- Container Runtime Interface (CRI): コンテナ オーケストレーション システムにおけるコンテナ ランタイムの構築方法に関する仕様
- Container Storage Interface (CSI): コンテナ オーケストレーション システムにおけるストレージ構築方法に関する仕様
- Service Mesh Interface (SMI): Kubernetesに重点をおいたコンテナ オーケストレーション システムにおけるサービス メッシュ構築方法に関する仕様

コンテナと
コンテナオーケストレーション

アプリケーション開発の変遷

物理サーバ上でアプリケーションを動かす

物理サーバ上で複数の仮想マシンを実行しその上でアプリケーションを動かす

クラウドベースのスケーラブルな仮想マシンでアプリケーションを動かす

さまざまなサーバをノードとして抽象化して
コンテナという単位でアプリケーションを動かす

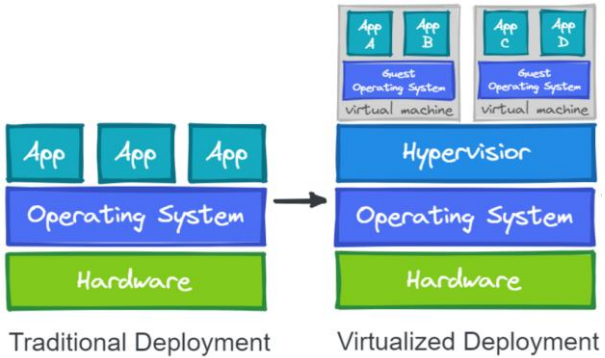
仮想マシンとは

- 物理的なサーバーと同等の機能・動作を再現した仮想的なハードウェア環境
- ハイパーバイザーと呼ばれるソフトウェアで実現
- ハイパーバイザーにはハードウェア上で直接動作する「Type1(ネイティブ型)」とハードウェアのホストOS上で稼働するアプリケーションが仮想マシンを再現する「Type2(ホスト型)」の2種類がある

Type1：KVM 、VMware ESX/ESXi、Hyper-V、Xenなど
Type2：VMware Workstation Player、VirtualBox、Microsoft Virtual Serverなど

仮想マシンの特徴

- 個別にOS(ゲストOS)、ライブラリ(実行環境)、ミドルウェア、アプリケーションをインストールして独立したコンピューティング環境が構築できる
- ハードウェアをエミュレートするため、CPUやメモリのリソースを多く消費する
- ホストOSとは独立しているため実行環境が変わってもそのままできる
- ホストOSとは異なる種類のOSを実行可能



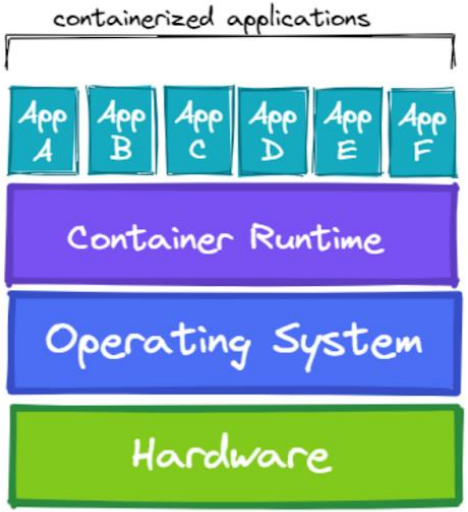
従来型アプリケーション開発

- OSやライブラリなどが整った物理・仮想マシンの実行環境で稼働
- 実行環境に依存するため、実行環境に差異があると障害が発生
- 実行環境のトラブルがアプリケーションに影響

- 仮想マシンはイメージの容量も多く、オーバーヘッドも大きい
- 仮想マシンイメージがベンダー特有
- OSやライブラリのアップデートがアプリケーションに影響

コンテナ

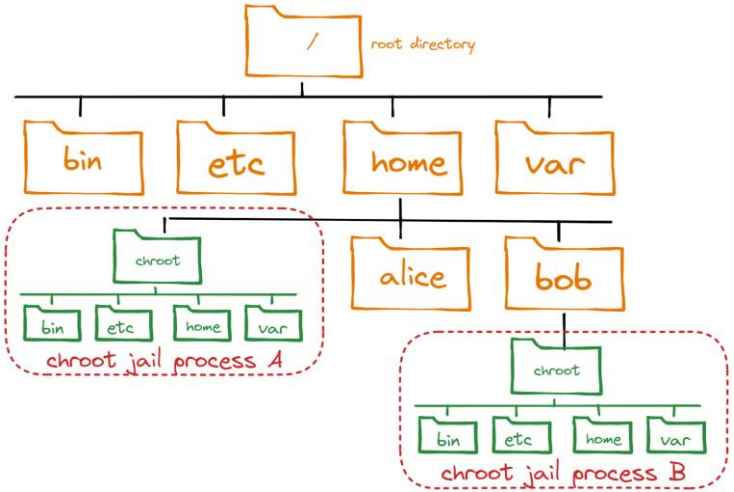
- 物理・仮想マシン上で稼働するホストOSのリソースの一部を隔離し、仮想的に作り出された実行環境。
- ホストOSとは隔離されたプロセスを作成し、その上にライブラリ、ミドルウェア、アプリケーションをインストールして、独立したコンピューティング環境を構築。
- ホストOSのプロセスとして動作するので、CPUやメモリのリソース消費は少なく、起動するまでの時間がかからない。
- コンテナランタイムがハードウェアやOSごとの違いを吸収するため、他の実行環境へコンテナを容易に移動・配布できる。
- アプリケーション単位の仮想化ともいえる



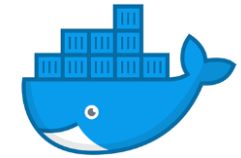
Container Deployment

コンテナ技術の背景

- プロセスをrootファイルシステムから分離させるUnixのchrootが祖先。
- 現在のLinux カーネルでは、様々なリソース分離のためにnamespaces（名前空間） 機能が提供されている。
  - pid - プロセスにそれ自身のプロセスIDセットを提供
  - net - プロセスにIPアドレスを含むそれ自身のネットワーク スタックを持つことを許可
  - mnt - ファイルシステムの表示を抽象化し、マウントポイントを管理
  - Ipc - 名前付き共有メモリ セグメントの分離を提供
  - user - ユーザーIDとグループIDセットをプロセスに提供
  - Uts - プロセスに自身のhostnameとdomainnameを持つことを許可
  - cgroup - プロセスにそれ自身のcgroupルート ディレクトリセットを持つことを許可
  - time - システムクロックの仮想化



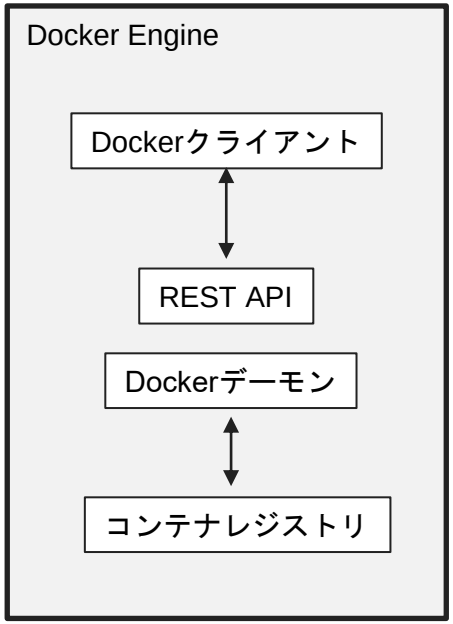
コンテナの代名詞「Docker」



- 各コンテナは、 Docker Engine(コンテナの動作プラットフォーム)上で動作。

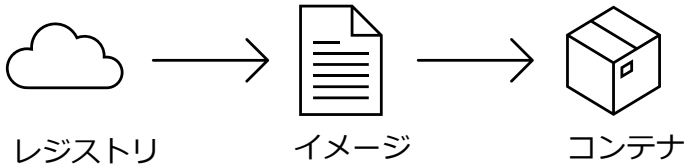
- Docker Engineは、コンテナやイメージを管理するためのアプリケーション。DockerクライアントからDockerデーモンのAPIにアクセスすることでコンテナに関するさまざまな操作が行える。

➢ コンテナを実行するためにDockerを使う必要はない。Open Container Initiative(OCI)は、コンテナ ランタイムのリファレンス実装「 runC」をメンテナンスしており、Dockerを含む様々なツールで使われている。



コンテナイメージとコンテナレジストリ

- 各コンテナはイメージと呼ばれるコンテナのテンプレートファイルから生成される。



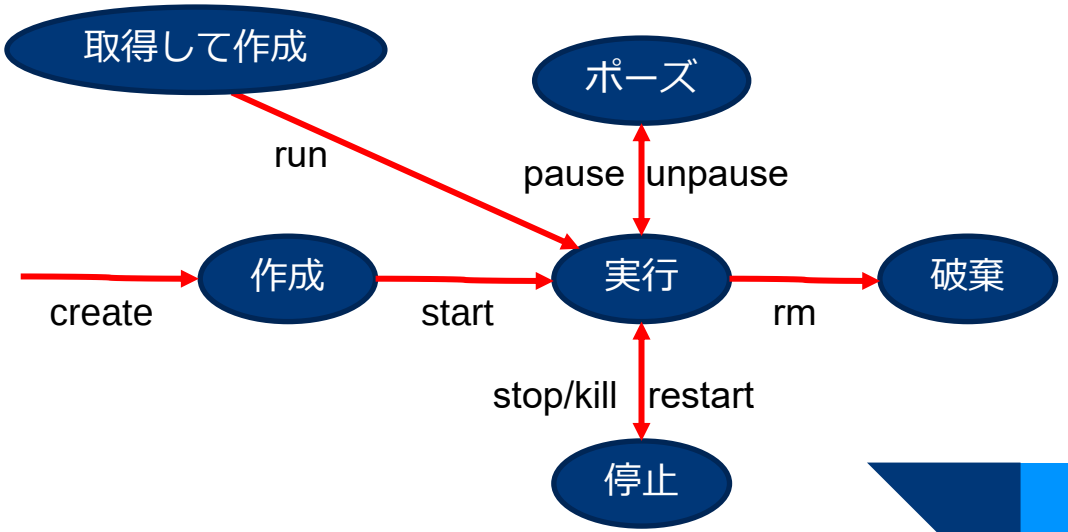
- ユーザーは独自にコンテナに含めるコンポーネントを定義してイメージを作成することもできるし、Docker Hubなどのレジストリなどで公開されている既成のイメージを取り込んでコンテナを生成することもできる。

コンテナのライフサイクル

- **作成　→　実行　→　(一時)停止　→　破棄**

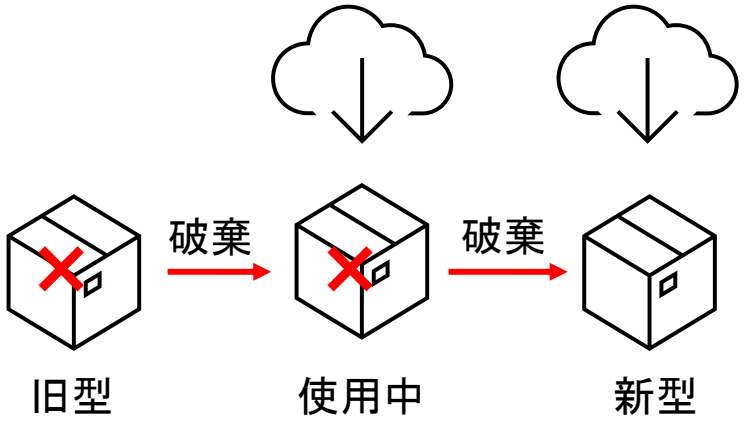
  という一連の流れ

- コマンドで状態が変化
  - 作成：docker create
  - 起動：docker start
  - 停止：docker stop
  - 破棄：docker rm



コンテナのライフサイクル

- 1つのコンテナをアップデートしながら使うのではなく、アップデートされた新しいコンテナを使う（＝次から次へとコンテナを乗り換える）。
- アップデートされたコンテナに乗り換える方が時間的にも速く間違いがない。



コンテナの実行（Dockerの場合）

 # docker　コマンド(上位・副)　オプション (オプションの引数)　対象　引数

オプション1　　対象

docker container run -it --name ubu1 ubuntu /bin/bash

上位コマンド　副コマンド　　オプション2(引数あり)　　引数

最新版のubuntuコンテナイメージを取得してubu1という名前で起動しbashターミナルを実行

主なDockerコマンド

主コマンド	上位コマンド	副コマンド	主なオプション	内容	旧体系(v.1.21以前)
docker	container	run	-d -i -t -p -v -e --name --rm	リポジトリからイメージをダウンロードしてコンテナを生成し起動する	docker run
			-d	コンソールを離してバックグラウンドで起動する（--detach）	
			-i	キーボードからの入力を標準入力としてシェルに伝える（--interactive）	
			-t	シェルのプロンプト表示を有効にする（--tty）	
			-p [ホストのポート番号:コンテナのポート番号]	ポート番号を指定する（--publish）	
			-v [ホストのディスク:コンテナのディレクトリ]	ボリュームをマウントする（--volume）	
			-e 環境変数=値	環境変数を指定する	
			--name [コンテナ名]	コンテナに名称を付与する	
			--rm	コンテナが停止されたらコンテナを即時破棄する	
		start	-i	コンテナを起動する	docker start
		restart		コンテナを再起動する	docker restart
		stop		コンテナを停止する	docker stop
		pause		コンテナを一時停止する	docker pause
		unpause		コンテナの一時停止を解除する	docker unpause
		kill		コンテナを強制終了する	docker kill
		attach		実行中のコンテナに接続する	docker attach
		create	-e -p -v --name	イメージからコンテナを作成する	docker create
		commit		コンテナをイメージに変換する	docker commit
		ls	-a	コンテナの一覧を表示する	docker ps
		rm	-f -v	停止済みのコンテナを削除する	docker rm
		prune		停止済みのコンテナをまとめて削除する	
		exec	-i -t	実行中のコンテナでプログラム（コマンド）を実行する	docker exec
		cp		コンテナとホスト間でファイルをコピーする	docker cp
		logs		コンテナのログを参照する	docker logs
		inspect		コンテナの詳細情報を表示する	docker inspect
		stats		コンテナのステータスを表示する	docker stats
docker	image	pull		リポジトリからイメージをダウンロードする	docker pull
		push		リポジトリにイメージをアップロードする	docker push
		tag		イメージにタグを付与する	docker tag
		ls		イメージの一覧を表示する	docker images
		rm		イメージを削除する	docker rmi
		build	-t	イメージを作成する	docker build
		inspect		イメージの詳細情報を表示する	

コンテナイメージのビルド

- コンテナイメージ

  アプリケーションを実行するために必要なもの全て（コード、ランタイム、システムツール、システムライブラリ 、設定など）が含まれた実行可能な軽量で独立型したソフトウェアのパッケージ。

  dockerの場合、Dockerfile から命令を読み込むことでイメージをビルドできる。命令はサーバーへアプリケーションをインストールするために使われるものとほとんど同じ。

- Dockerfile

  カスタマイズしたコンテナイメージを作成するための手順書のようなもの。

コンテナの特徴（まとめ）

- 1台のホストOSに複数のサーバー(アプリケーション)を構築できる
  （同じ種類のサーバーの複数構築も可能）
- 動作が軽く、データ容量も軽量
- 互いに隔離されている＝安全で管理もしやすい
- 作成、複製、破棄、アップデート、入れ替えなどが容易
- カスタマイズしたコンテナをイメージ化して配布できる
- 開発環境や運用環境といった異なる環境で同じコンテナを再現できる
  （可搬性に優れる）
- 物理マシンに問題が起こると構築したコンテナ全てに影響が出る
- Linuxとコンテナエンジンが必須

コンテナの問題

- 分離し自立した小さいコンテナの疎結合はマイクロ サービス アーキテクチャの基本。
- コンテナは、アプリケーションを集約して実行する良い方法だが、ダウンタイムが発生しないように、コンテナを管理する必要がある。
- 大量のコンテナをデプロイ、管理しなければならない場合、セキュリティやネットワーキング、ストレージ、監視、サービスディスカバリのアプローチなどで解決すべき問題がでてくる。

コンテナ オーケストレーション

- コンテナ オーケストレーション システムは、複数サーバーのクラスタを構築し、そこにコンテナをホストする方法を提供。

- 代表的なシステムがKubernetes。

kubernetes

kubernetesとは

Kubernetesとは

- Kubernetesは、オープンソースのコンテナ オーケストレーション プラットフォーム。コンテナ化したワークロードのデプロイやスケーリング、管理ができる。
- コンテナオーケストレーションの標準システムとして選ばれている。
- コンテナ ランタイム、モニタリング、アプリケーション デリバリー ツールなど、様々なクラウド ネイティブ技術が周辺で発展。
- 複数のデータセンターやリージョンにまたがる、数千ものサーバーノードのクラスタを扱うことが可能。

Kubernetesとは

- ギリシャ語に由来。操舵手やパイロットの意味。
- K8sが略語。“K”と“s”の間に8文字あるから。
- Googleが開発。2014年にKubernetesプロジェクトをCNCFに寄贈しオープンソース化。
- Kubernetesの起源に関するドキュメンタリー
  - https://youtu.be/BE77h7dmoQU

Kubernetesが提供するもの

- **サービスディスカバリーと負荷分散**

  DNS名または独自のIPアドレスを使ってコンテナを公開。コンテナへのトラフィックが多い場合は、ネットワークトラフィックを振り分けることが可能。

- **ストレージ オーケストレーション**

  ローカルやパブリッククラウドなど、選択したストレージシステムを自動でマウント可能。

- **自動化されたロールアウトとロールバック**

  コンテナのあるべき状態を記述することができ、制御されたスピードで実際の状態をあるべき状態に変更可能。

https://kubernetes.io/ja/docs/concepts/overview/

Kubernetesが提供するもの

- **自動ビンパッキング**

  コンテナ化されたタスクを実行するノードのクラスターを提供。各コンテナがどれくらいCPUやメモリーを必要とするのかを宣言。コンテナをノードにあわせて調整可能。

- **自己修復**

  処理が失敗したコンテナを再起動し、コンテナを入れ替え、定義したヘルスチェックに応答しないコンテナを強制終了。

- **機密情報と構成管理**

  機密情報を保持、管理。機密情報をデプロイし、コンテナイメージを再作成することなくアプリケーションの構成情報を更新。

https://kubernetes.io/ja/docs/concepts/overview/

Kubernetesアーキテクチャ

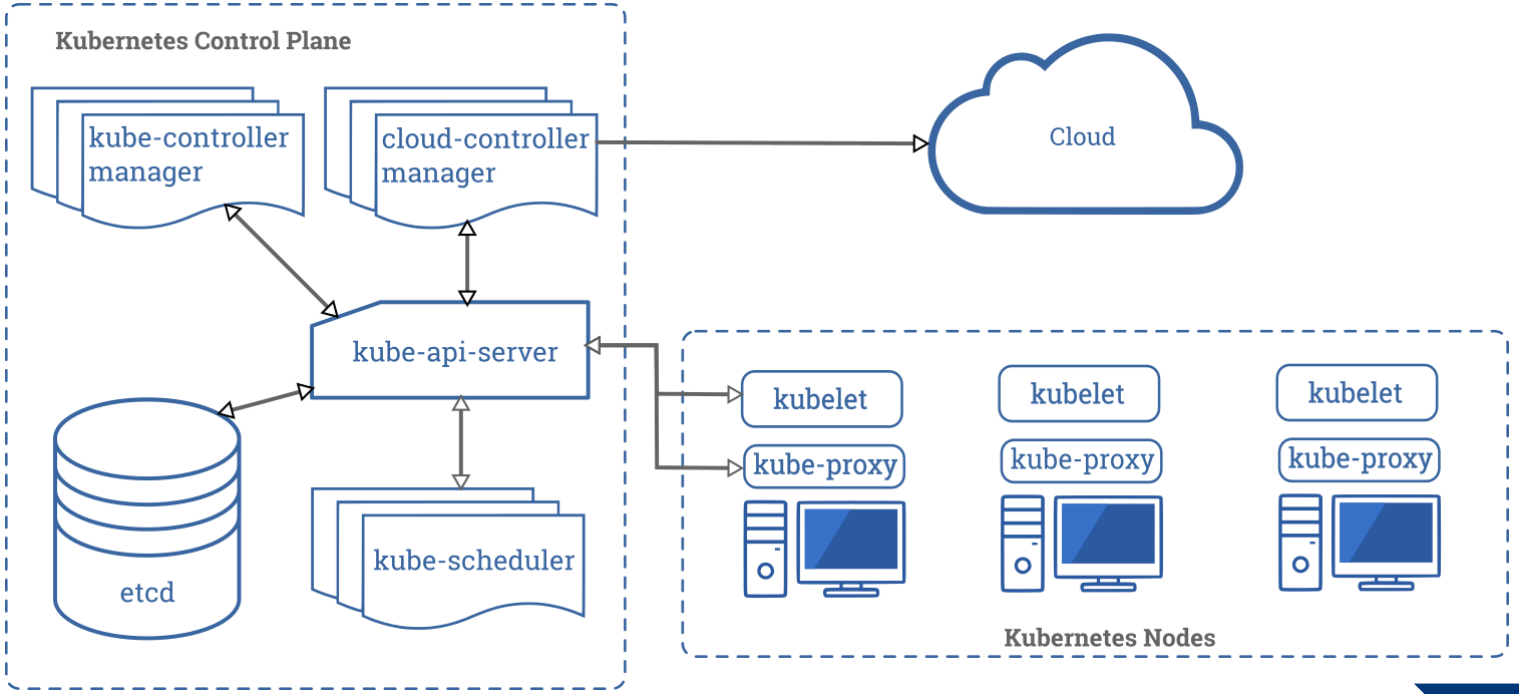
Kubernetesクラスターは、 コンテナ化されたアプリケーションを実行するノードと呼ばれるワーカーマシンの集合。

- コントロール プレーン ノード／Control plane node(s)
  - クラスター内のワーカーノードとPodを管理
  - 複数のマスターノードを使用して、クラスターにフェイルオーバーと高可用性を提供
- ワーカー ノード／Worker nodes
  - アプリケーションのコンポーネントであるPodをホスト

本番環境では、コントロールプレーンは複数のコンピューターを使用し、クラスターは複数のノードを使用し、耐障害性や高可用性を提供

https://kubernetes.io/ja/docs/concepts/overview/components/

Kubernetesアーキテクチャ



コントロール プレーン コンポーネント

- **kube-apiserver**

  Kubernetes APIを外部に提供するコンポーネントでコントロールプレーンのフロントエンド。ユーザーやサービスは、APIを通してKubernetesに属するリソースを作成、修正、削除、検索できる。リクエストに対する認証・認可・入力制御の役割も備えている。

- **Etcd**

  一貫性、高可用性を持ったキーバリューストアで、Kubernetesの全てのクラスター情報の保存場所。

- **kube-scheduler**

  新しく作られたPodにノードが割り当てられているか監視し、割り当てられていなかった場合にそのPodを実行するノードを選択。

コントロール プレーン コンポーネント

- **kube-controller-manager**

  ノードがダウンした場合の通知と対応を担当するノードコントローラー。
  Jobオブジェクトを監視し、実行して完了させるためのPodを作成するコントローラー、ServiceとPodを紐付けるコントローラー、新規の名前空間に対して、デフォルトのServiceAccountを作成するコントローラーなど、複数のコントローラープロセスを実行。

- **cloud-controller-manager**

  Kubernetesとパブリッククラウド間を中継。Kubenetesで使いたいリソースとクラウドのリソースを連携。

ワーカーノード コンポーネント

- **コンテナランタイム**

  コンテナの実行を担当するソフトウェア。 高レベルのcontainerdやCRI-Oや低レベルのruncやgVisorなど、複数のコンテナランタイムをサポート。高レベルランタイムからの情報をもとに低レベルランタイムがコンテナ環境を作成。

- **Kubelet**

  クラスター内の各ノードで実行されるエージェントでPodの起動や管理を担う。Schedulerからの指示を受け取り、コンテナランタイムを操作してPodを作成。

- **kube-proxy**

  クラスター内の各ノードで動作しているネットワークプロキシ。役割はネットワーキングと負荷分散のサポート。Linuxのiptablesを使ってPodあての通信をPodへ転送するための通信制御を担う。KubernetesのServiceの機能の一部を実装。

Kubernetesの代表的なオブジェクト

- **Pod**

  Kubernetesオブジェクトの最小単位。固有の仮想IPアドレスが割り振られ、Pod内のコンテナはIPアドレス、ポート、名前空間、ボリュームを共有し、Pod間の通信が可能。

- **ReplicaSet**

  希望数のPodが動作していることを保証（あるべき姿を維持）するコントローラ オブジェクト。Podのコピーを複数起動することでアプリケーションのスケールアウトを実現。

- **Deployment**

  ReplicaSetを管理するオブジェクト。アプリケーション全体のローリングアップデートやロールバックを実現。

Kubernetesの代表的なオブジェクト

- **Service**

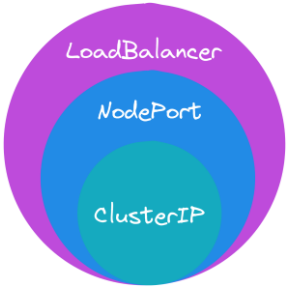
  Kubernetesのクラスタ内外からのPodあて通信を仮想IPアドレスで振り分ける。
  ClusterIP、NodePort、LoadBalancer、ExternalIPなどのサービスタイプがある。

- **PV/PVC**

  PersistentVolume(PV)はPodのデータを保存する永続ストレージ。
  PersistentVolumeClaim(PVC)は必要なストレージ容量を動的に確保する抽象化ストレージ。

- **ConfigMap**

  Podやコンテナの設定値（環境変数、ポートなど）や設定ファイルを保存。



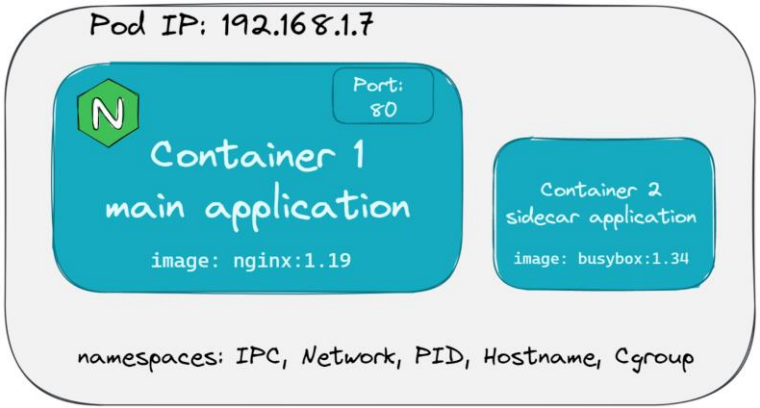
Kubernetesでのコンテナの実行

- Kubernetesでは、コンテナを直接起動せず、最小単位としてPodを定義し、Pod上でコンテナを実行する。

- Podオブジェクトを作成すると、ノードを実行するコンテナを取得するまで、複数のコンポーネントがそのプロセスに携わる。

Kubernetesでのコンテナの実行

- 各種ワークロード情報をYAML形式でマニフェスト（定義）ファイルに記述する。




```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-sidecar
spec:
  containers:
  - name: nginx
    image: nginx:1.19
    ports:
    - containerPort: 80
  - name: count
    image: busybox:1.34
    args: [/bin/sh, -c,
            'i=0; while true; do echo
"$i: $(date)"; i=$((i+1)); sleep 1;
done']
```

マニフェスト例

hoge.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.19
        ports:
        - containerPort: 80
```

必須フィールド

- apiVersion
  APIグループのバージョン管理。
- kind
  作成するオブジェクトの種類。
- metadata
  リソースの名前やラベル情報。認証のために使われる。
- spec
  リソースの内容や仕様。あるべき状態を記述。

Kubernetesでの通信

- APIへのアクセスには、コマンドライン インタフェース クライアントの kubectl を使う。
- YAMLファイルからKubernetesでオブジェクトを作る場合。-fオプションの後にyamlファイルを指定。

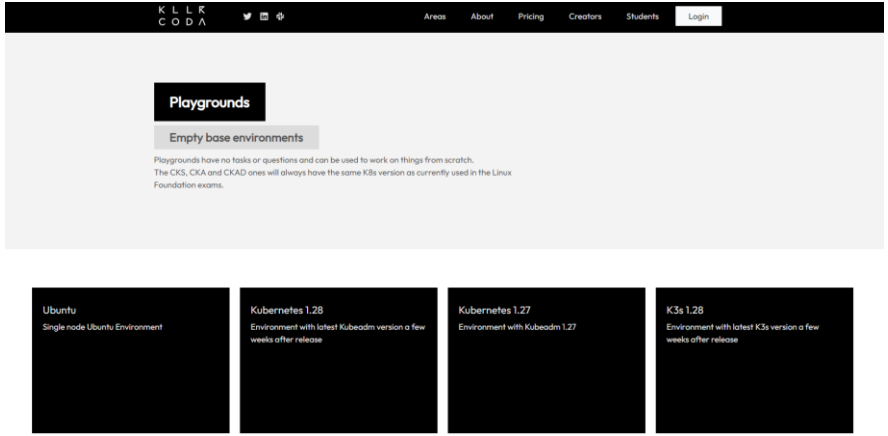
  $ kubectl create -f hoge.yaml

- 状態確認

  $ kubectl get deployment

Kubectlコマンドの実行例

- Killercodaのプレイグラウンドを利用
- Killercodaの特徴
  - ワンクリックでLinux環境が使える
  - Kubernetesの学習用コンテンツが充実
  - 学習用コンテンツの自作や公開も可能
  - 各種制限がある
- コマンド実行例
  - kubectl get node
  - kubectl get pods –A
  - kubectl create –f hoge.yaml
  - kubectl get deployment
  - kubectl describe deploy nginx-deployment
  - kubectl get pod –o wide



https://killercoda.com/

Kubernetesを学ぶには

- Kubernetes公式
  - https://kubernetes.io/ja/training/
- Linux Foundationのトレーニング
  - Kubernetes 基礎 (LFS258-JP)
    https://training.linuxfoundation.org/ja/training/kubernetes-fundamentals-lfs258-jp/
  - Kuberbetesとクラウドネイティブ基礎 (LFS250-JP)
    https://training.linuxfoundation.org/ja/training/kubernetes-and-cloud-native-essentials-lfs250-jp/
- Udemy
  - Certified Kubernetes Administrator (CKA) with Practice Tests
    https://www.udemy.com/course/certified-kubernetes-administrator-with-practice-tests/
  - Kubernetes Certified Application Developer (CKAD) with Tests
    https://www.udemy.com/course/certified-kubernetes-application-developer/
- 書籍
  - Kubernetes完全ガイド 第2版
  - Dockerから入るKubernetes コンテナ開発からK8s本番運用まで

THE LINUX FOUNDATION

Kubernetes関連資格

- 認定Kubernetesクラウドネイティブアソシエイト（KCNA-JP）
  https://training.linuxfoundation.org/ja/certification/kubernetes-and-cloud-native-associate-kcna-jp/

  ※受験準備にはLFS250-JPのトレーニング受講がおすすめ。

- 認定 Kubernetes 技術者 (CKA-JP)
  https://training.linuxfoundation.org/ja/certification/certified-kubernetes-administrator-cka-jp/

  ※受験準備にはLFS258-JPのトレーニング受講がおすすめ。

- Kubernetes アプリケーション認定開発者 (CKAD-JP)
  https://training.linuxfoundation.org/ja/certification/certified-kubernetes-application-developer-ckad-jp/

- Kubernetes 認定セキュリティスペシャリスト (CKS-JP)
  https://training.linuxfoundation.org/ja/certification/certified-kubernetes-security-specialist-jp/

THE LINUX FOUNDATION

お問い合わせ先

- **LPI-Japan**　(Linux Foundation認定販売代理店)
  https://lpi.or.jp/k8s/

- **Linux Foundation**
  https://onl.la/uR1SaLz

THE LINUX FOUNDATION

最後に

- 関連資格受験の感想
- ご清聴ありがとうございました


THE
LINUX
FOUNDATION