Engineer

コンテナーランタイム

コンテナーイメージから隔離されたプロセスを起動し、その実行を管理するソフトウェア

どのようなものか

コンテナーランタイムは、ディスク上のコンテナーイメージと、カーネル上で隔離されて動くプロセスの間にあるソフトウェアです。この言葉は二つの層を指し、人によってどちらを意味するかが異なります。OCIランタイムとも呼ぶ低レベルのランタイムと、コンテナーエンジンと呼ぶことの多い高レベルのランタイムです。

低レベルのランタイムは、限られた一つの役割を果たします。バンドルを読み、設定に記述された名前空間、cgroups、マウントをカーネルに要求して、その中でプロセスを起動します。Open Container Initiativeのランタイム仕様は、バンドルを、`config.json`ファイルとコンテナーのルートファイルシステムを含むディレクトリーと定義しています。仕様は、すべてのランタイムが対応すべき操作としてstate、create、start、kill、deleteを定めていますが、コマンドラインの形式は各実装に任せています。バージョン1.0は2017年7月に公開されました。OCI自身が保守するruncは、自らを「エンドユーザーを想定して設計されたものではない、低レベルのツール」と説明しています。crunはCで実装され、より少ないメモリーで同じ仕事をします。Podmanのドキュメントでは既定のランタイムとされています。三つ目のyoukiは、Rustで書かれています。

その上にエンジンがあります。containerdは、自分自身を、ホスト上のコンテナーのライフサイクル全体を管理するデーモンと説明しています。イメージの転送と保管、コンテナーの実行と監督、それらに伴うストレージとネットワークの接続を扱います。CRI-Oも似た仕事をしますが、Kubernetes専用です。エンジンはイメージを取得して展開し、バンドルを書き出して低レベルのランタイムに渡し、そのランタイムが終了した後もプロセスの監視を続けます。Docker Engineは、さらに一段上にあります。そのドキュメントによると、コンテナーのライフサイクルにはcontainerdを使い、containerdは既定でruncを使います。Podmanは別の方法を取り、常駐デーモンを持ちません。各コマンドが自らバンドルを用意してcrunかruncを呼び、稼働中のコンテナーの監視はconmonという小さなプログラムに任せます。

Kubernetesでは、さらに一つの用語が加わります。各ノードのkubeletは、Container Runtime Interface、略してCRIを介してエンジンと通信します。これは、containerdとCRI-Oの両方が実装するgRPCプロトコルです。Docker EngineはCRIより前から存在するため、Kubernetesは長年、dockershimというアダプターを組み込んでいましたが、v1.24で削除しました。

概念を判断に生かす

「コンテナーランタイムとは？」を実際の判断に生かしましょう。その概念を、開発または購入するシステムに結び付けて考えるお手伝いをします。 ノートパソコン、テストサーバー、本番環境で同じ動作になるようにまとめ、忙しい週には処理能力を増やせるようにします。

サービスを見る お問い合わせ →

用語の背後にある前提を検証する

解説とリンク先の実績を参考に、プロジェクトについての質問を整理してください。ツールを選ぶ前に、前提を一緒に検討できます。 Google Cloud・AWSでDockerアプリ20個を運用 — 20 デプロイし監視したコンテナアプリケーション

Google Cloud・AWSでDockerアプリ20個を運用 お問い合わせ →

なぜ重要なのか

コンテナーランタイムを選ぶことは、実際にはどの層を変えるかを選ぶことです。二つの層では、得られるものが異なります。

エンジンは、コンテナーの日常的な運用の仕組みを決めます。誰がイメージを取得して保管し、異常終了したコンテナーを再起動するか、ログをどこへ送るか、ほかのツールがどのAPIを使い、誰がプロセスツリーを管理するかです。サーバー上でDocker EngineをPodmanに替えたり、Kubernetesノード上でcontainerdに替えたりすると、これらはすべて変わります。一方、コンテナー自体はそのままです。基盤となるイメージ形式とバンドルが、共通のOCI標準に従うためです。dockershimの削除に関するKubernetesのドキュメントも、明確に述べています。`docker build` で生成したイメージは、すべてのCRI実装で動きます。

低レベルのランタイムは、プロセスをどの程度隔離するかを決めます。runcとcrunは、ホストのカーネル上で通常のプロセスを起動します。高速で、完全な互換性もありますが、同じマシンのすべてのコンテナーが、一つのカーネルを共有することになります。gVisorのランタイムrunscは、Goで書かれたアプリケーションカーネルをコンテナーとホストの間に置き、コンテナーのシステムコールに自ら応答します。そのドキュメントは、代償として、アプリケーションの互換性が低下し、システムコールごとの負担が増えるとしています。Kata Containersは、専用のカーネルを持つ軽量な仮想マシン内で各処理を起動し、ハードウェア仮想化を第二の防御層にします。実行する各ノードには、ハイパーバイザーが必要です。

Kubernetesでは、二つ目の選択をクラスター単位ではなくPod単位で行えます。RuntimeClassは、エンジンに設定されたハンドラーを指定します。`runtimeClassName` を設定したPodはそのハンドラーで動くため、信頼できない処理だけをgVisorやKataに置き、残りはruncのままにできます。ドキュメントは、これを性能とセキュリティの釣り合いの問題として説明しています。

実際の活用例

日常の会話では、この言葉はエンジンの層を指すようになりがちです。このサイトで紹介する仕事も、その意味で使っています。

Dockerでパッケージ化し、二つのクラウドと少数のコンテナーランタイムに分散していた二十のアプリケーションは、一貫した方法でデプロイでき、問題が起きたらすぐに原因を調べられる必要がありました。パッケージ化にはDockerを使い、問題の調査・対処にはPodmanを、オーケストレーションのデバッグにはKubernetesを使いました。このようにエンジンをまたいで作業できるのは、先に説明した層分けによります。エンジンは異なっていても、その下のOCIのイメージ標準とランタイム標準は共通しています。

新しいホストでは、エンジンの層で選択しました。コンテナー基盤はDockerの代わりにPodmanとQuadletで動きます。既定のエンジンが、強化用ロールの設けるファイアウォールルールより優先される形でコンテナーのポートを公開することと、systemdの隣にもう一つの監督役を追加することが理由です。この選択で変わったのは、誰が各コンテナーを起動し、その後の監視を担うかでした。コンテナーをサービスとして扱い、ホストが既に信頼する監督役の下で、通常のサービスと同じように起動するようにしました。

同じ考え方は、当社の公開サービスを載せる512 MBのホストにも当てはまります。そこでは、メモリーがすべての判断を決めました。使用可能なメモリーが約464 MBのマシンで選んだのは、独自の監督機構を持つデーモンではなく、systemdの下で動くQuadletユニットによるコンテナー基盤です。

問題が起きるところ

ランタイムの問題は、層と層の境目で表面化しがちです。互いに、相手は自分の期待どおりに設定されているはずだと考えるためです。

クラスターを更新すると、ノード上のツールからコンテナーが見えなくなります。 Kubernetes v1.24以降、kubeletはcontainerdかCRI-Oと直接通信するため、クラスターが配置するコンテナーはDocker Engineを通りません。ノード上の `docker ps` が表示するのは、Docker自身が起動したものだけです。Docker CLIを呼ぶか `docker.sock` をマウントする監視エージェント、起動スクリプト、特権Podは、それまでのように状況を把握できなくなります。処理は動き続けます。`crictl` やエンジン自身のAPIに移す必要があるのは、その周囲のツールです。

kubeletとランタイムが、別々の方法でcgroupsを管理します。 両者は、`cgroupfs` か `systemd` という同じcgroupドライバーを使う必要があり、kubeletの既定は `cgroupfs` です。initシステムにsystemdを使うホストでは、同じリソースについて異なる見方を持つ二つのcgroup管理機構が存在することになります。Kubernetesのドキュメントは、このようなノードはリソースが逼迫すると不安定になる場合があると注意しています。対策は、両方をsystemdドライバーにすることです。cgroup v2でも、このドライバーが推奨されます。containerd 2.xとruncでは、ランタイム側の設定は次のようになります。

```
# /etc/containerd/config.toml, containerd 2.x
[plugins.'io.containerd.cri.v1.runtime'.containerd.runtimes.runc.options]
  SystemdCgroup = true
```

kubelet側は、その設定に `cgroupDriver: systemd` を指定します。その後、containerdの再起動も必要です。containerd 1.xではセクションの書き方が異なり、Kubernetesのページには両方の形式が載っています。既にクラスターへ参加したノードでドライバーを変えるのは、慎重さを要します。一方のドライバーで作成したPodを、もう一方で再作成すると、エラーになる場合があるためです。したがって、ドキュメントでは、新しい設定でノードを置き換えるか再インストールするよう勧めています。

サンドボックス化したランタイムに移したPodが、起動時に止まります。 RuntimeClassが指定するハンドラーをノードのエンジンが持たない場合、Podは直ちにFailedフェーズへ入り、理由はイベントに記録されます。その条件を満たしても、処理自体がサンドボックスに適していなければなりません。gVisorは自分のカーネルでシステムコールに応答するため、あまり一般的でないカーネル機能に依存するソフトウェアは、動作が変わることがあります。ドキュメントも、システムコールの多い処理は適さないとしています。Kataにはノード上のハードウェア仮想化が必要で、クラウドの仮想マシンでは、これは入れ子の仮想化を意味します。RuntimeClassの `scheduling` にノードセレクターを指定すれば、実行可能なノードだけに、そのようなPodを配置できます。

手動で取り込んだイメージがkubeletから見えません。 containerdは、イメージ名とメタデータを名前空間ごとに保持します。`ctr` クライアントは、指定がなければ `default` 名前空間で動きますが、kubeletが通信するCRIプラグインは `k8s.io` を使います。そのため、単に `ctr images import` で取り込むと、kubeletが見ない場所に入り、Podはレジストリーから取得し直すか、イメージ取得エラーで停止します。`ctr -n k8s.io images import` で取り込み、kubeletと同じものが見える `crictl images` で確認すれば、この食い違いを避けられます。

参考資料

- OCIランタイム仕様 — すべての低レベルランタイムが実装する標準です。バンドル、`config.json`、ライフサイクルの操作を定めます。
- runc — OCI自身のランタイムです。上位のツールがどう操作するかの説明もあります。
- コンテナーランタイム — cgroupドライバー、CRIのバージョン、Kubernetesが求めるcontainerdとCRI-Oの設定を説明します。
- Container Runtime Interface（CRI） — kubeletとエンジンの間のgRPCプロトコルを説明します。
- Runtime Class — Podごとに異なるランタイムハンドラーを選び、対応するノードだけに配置する方法を説明します。
- 代替コンテナーランタイム — Docker Engineがコンテナーを gVisor、Kata Containers、youki、Wasmtimeに渡す方法を説明します。

用語集

実際の制約に合わせて選ぶ

デプロイの流れ、運用上のリスク、緊急対応の担当者をお知らせください。検査、復旧手段、明確な責任の所在を伴う変更の範囲を定められます。

料金を見る お問い合わせ →

用語を実務に結び付ける

「コンテナーランタイムとは？」がお客様の組織にどう当てはまるかをご相談ください。現在のシステムの具体例が一つあれば、話を始められます。

サービスを見る

welcome@engineer.company +45 50 10 61 84 デンマーク、コペンハーゲン

このページを持ち歩く

連絡先カード

心を込めて丁寧に制作： D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

著作権 © 2025 – 現在 · Engineer ApS · すべての権利を留保します。

印刷元： https://engineer.company/ja/glossary/container-runtime/