

Appendix

本書の付録では、読み進めていく過程でカバーしきれなかった基礎技術や関連情報を補足し、読者様がより深い理解を得られるように構成しています。特にOCIやクラウドネイティブ技術において、知識のギャップを埋めることを目的としています。



Kubernetes の基礎知識

Kubernetes のアーキテクチャ

Kubernetes のアーキテクチャは、クラスタ全体を管理する Control Plane (コントロールプレーン) と、アプリケーションを稼働させる Node (ノード) に大きく分けられます (図-01)。

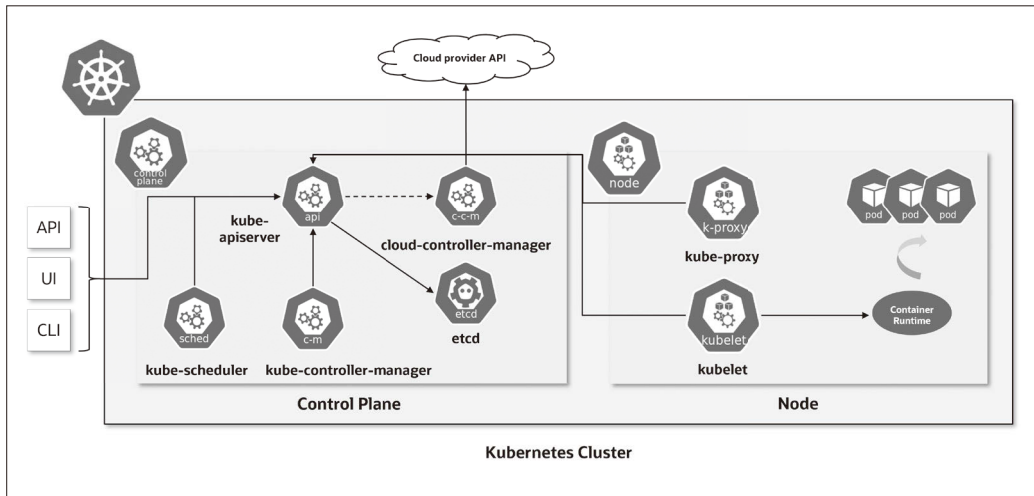


図-01 Control PlaneとNode

Control Planeはクラスタ全体の状態を監視し、望ましい状態を維持するためにNodeに指示を出します。一方、Nodeはその指示に基づいてアプリケーションを稼働させ、状態をControl Planeに報告します。それぞれの主要コンポーネントを説明します。

≫ Control Plane (コントロールプレーン)

Control Planeは、Kubernetesクラスタ全体を管理および制御するための中枢です (図-02)。

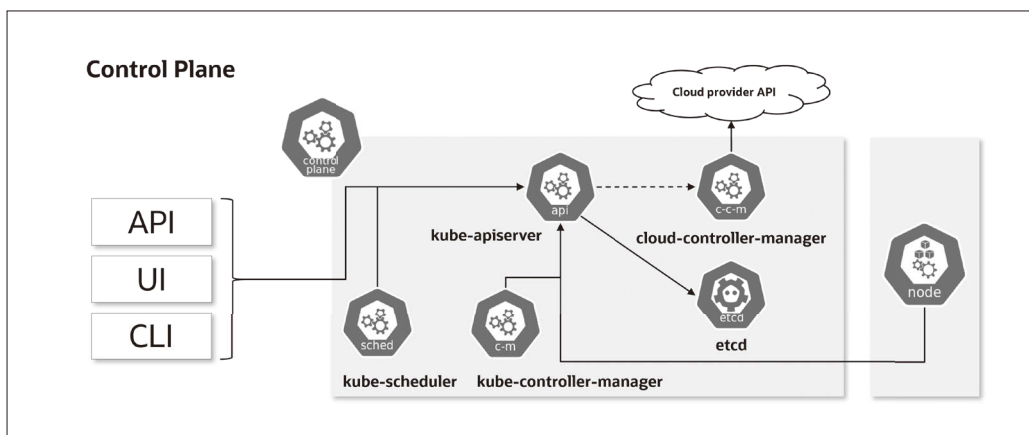


図-02 Control Plane

コントロールプレーンの主要コンポーネントを表-01に整理します。

主要コンポーネント	概要
APIサーバ (kube-apiserver)	Kubernetes クラスタの中核となるコンポーネントで、全ての操作がこのAPIを通じて実行される。kubectl コマンドや外部アプリケーションからのリクエストを受け付け、認証および認可を実施する。
スケジューラー (kube-scheduler)	新しいPodをどのNodeに配置するかを決定する。Nodeのリソース使用状況やポリシーを考慮し、最適なNodeを選定する。
コントローラーマネージャー (kube-controller-manager)	クラスタの状態を監視し、望ましい状態を維持するために動作するコントローラー群を統括する。例えば、Deploymentのレプリカ数の維持やNodeの監視を行う。
クラウドコントローラーマネージャー (cloud-controller-manager)	Kubernetes をクラウドサービスプロバイダに統合する役割を担う。クラウドのリソース (例：ロードバランサー、ボリューム、ノード) を管理するためのコントローラーを提供。クラウドサービスプロバイダに依存する処理がここで実行される。
etcd	Kubernetes クラスタ全体の状態を保存する分散型キー・バリューストア。クラスタ内の構成情報やリソースの情報が保存され、全コンポーネントがこのデータを共有する。

※ Pod、Deploymentについては、後述します。

表-01 Kubernetes Control Planeの主要コンポーネント

>> Node (ノード)

Nodeは、実際にアプリケーションのコンテナが稼働する実行環境です (図-03)。1つのクラスタには複数のNodeが含まれます。

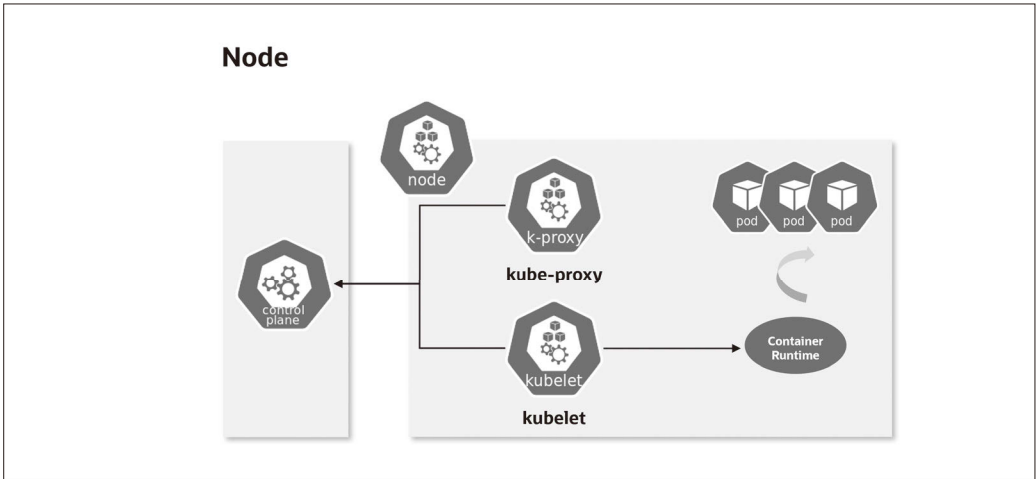


図-03 Node

Nodeの主要コンポーネントを表-02に整理します。

主要コンポーネント	概要
kubelet	Control Planeからの指示を受け、Podのライフサイクル（作成、更新、削除）を管理する。
kube-proxy	クラスタ内のネットワーク通信を管理する。サービスディスカバリーや負荷分散の設定を行い、Pod間や外部との通信を可能にする。
コンテナランタイム (Container Runtime)	Pod内のコンテナを実際に行うためのランタイム。KubernetesはCRI (Container Runtime Interface) を通じてコンテナランタイムと通信をする。OKEのコンテナランタイムはCRI-O ^{*1} 。

表-02 Kubernetes Nodeの主要コンポーネント

>> Control Plane と Node の関係

Control PlaneとNodeの各コンポーネントは、APIサーバを経由してetcdにある情報を確認および更新してクラスタの状態を自立的に維持します。Podが作成されるまでの仕組みを例に各コンポーネントの動作を理解します（図-04）。

※1 CRI-Oは、Kubernetesに特化した軽量なコンテナランタイムで、OCI (Open Container Initiative) 規格に準拠したコンテナイメージやランタイムをサポート。公式ウェブサイト：<https://cri-o.io/>

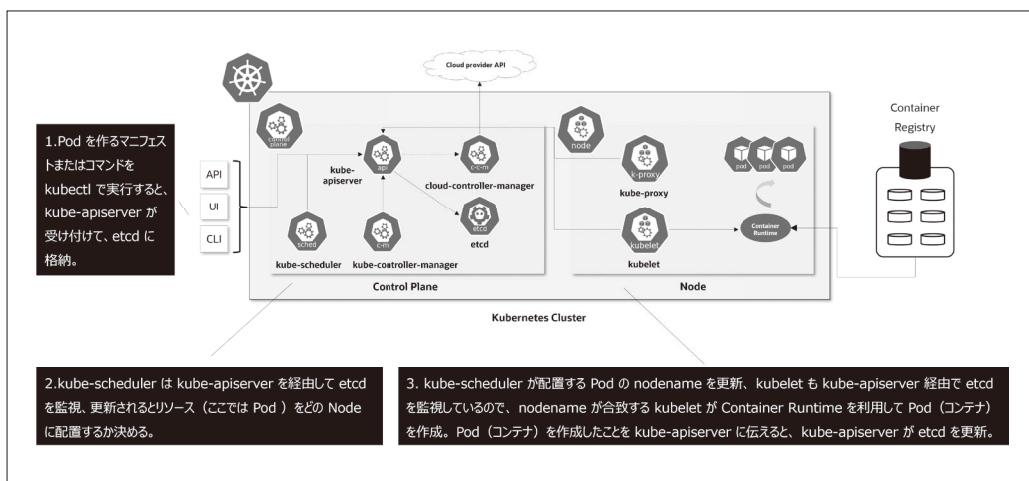


図-04 Podが作成されるまでの仕組み

Kubernetes のリソースとオブジェクト

Kubernetes には、リソースとオブジェクトという仕組みがあります。

Kubernetes クラスタ内でアプリケーションを動かしたり、設定を管理するために「リソース」を使用します。リソースは、Kubernetes が提供する標準機能で、Pod、ReplicaSet、Deployment、Service、Ingress、PersistentVolume、PersistentVolumeClaim、ConfigMap、Secret、Namespace などが含まれます。リソースは「何を実行したいか」を示す設計図のようなものです。

一方、オブジェクトは、そのリソースを基に作られる実体で、クラスタ内で実際に動作します。例えば、Pod リソースを定義したマニフェストを適用することで、Pod オブジェクトがクラスタ内に作成され、アプリケーションが動作します。

さらに、Kubernetes では標準リソースだけでなく、独自のリソースを追加することもできます。これをカスタムリソース (Custom Resource) と呼びます。カスタムリソースは、特定のアプリケーションやツールに合わせた独自の機能を拡張するために使用されます。例えば、CI/CD のパイプライン管理やデータベースの自動化などに活用されます。カスタムリソースを利用する際には、CustomResourceDefinition (CRD) を作成してクラスタに新しいリソースタイプを登録します。

Kubernetes の代表的なリソースとオブジェクトである Pod、ReplicaSet、Deployment、Service、Ingress、PersistentVolume、PersistentVolumeClaim、ConfigMap、Secret、

Namespaceについて概要を説明します。

>> Pod

Podは、Kubernetesクラスタ内でアプリケーションのコンテナを実行および管理する最小単位です。1つのPodには1つのコンテナが含まれることが多いですが、複数のコンテナを含むことで密接に連携した動作も可能です。Podの特徴を表-03に整理します。Podの概要図を図-05に示します。

特徴	概要
ネットワーク	Pod内のコンテナは、同じIPアドレスとホスト名を共有するが、同じポート番号を同時に持つことはできない（ポート番号はPod内で一意）。Pod内のコンテナ間はlocalhost (127.0.0.1) を使って通信できる。
ストレージ	ボリュームを使ってPod内のコンテナ間でデータを共有。
管理対象	ReplicaSet、Deploymentなどの上位リソースによって管理される。

表 -03 Podの特徴

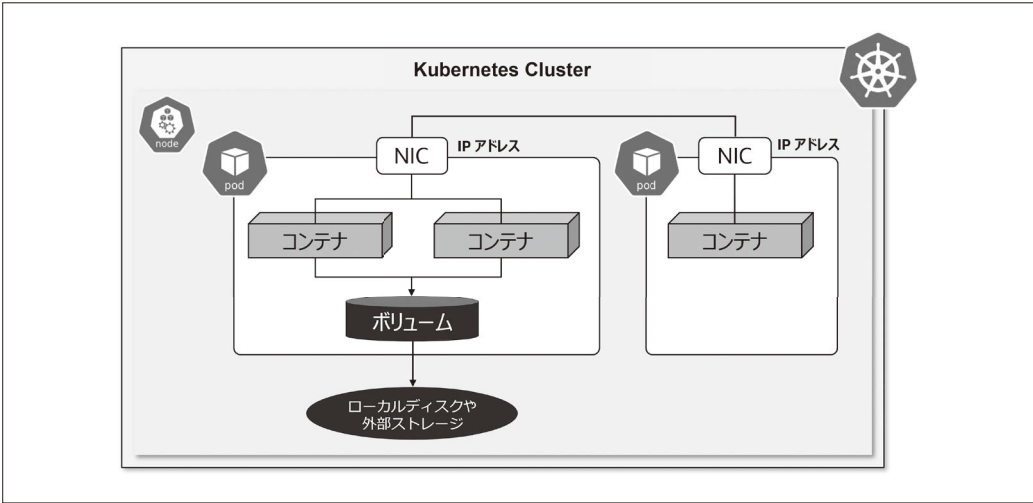


図 -05 Podの概要図

Pod内のコンテナをどのように組み合わせるかを設計する際に、参考となるデザインパターンがあります。これらのパターンを活用することで、各コンテナの役割を明確にし、効率的で柔軟な構成を実現できます。主なデザインパターンを表-04に整理します。各デザインパターンの用途例のイメージが図-06です。

パターン名	概要	主な用途例
サイドカーパターン	メインコンテナの機能を補助するコンテナを追加するパターン。	メインコンテナから出力されたログを外部ストレージに転送。
アンバサダーパターン	外部システムとの連携を代行するコンテナを追加するパターン。	環境が異なるデータベースへの動的アクセスを管理。
アダプターパターン	外部からのアクセスを調整するコンテナを配置するパターン。	監視アプリケーション用のメトリクスを専用フォーマットに変換して提供。

表-04 主なPodのデザインパターン

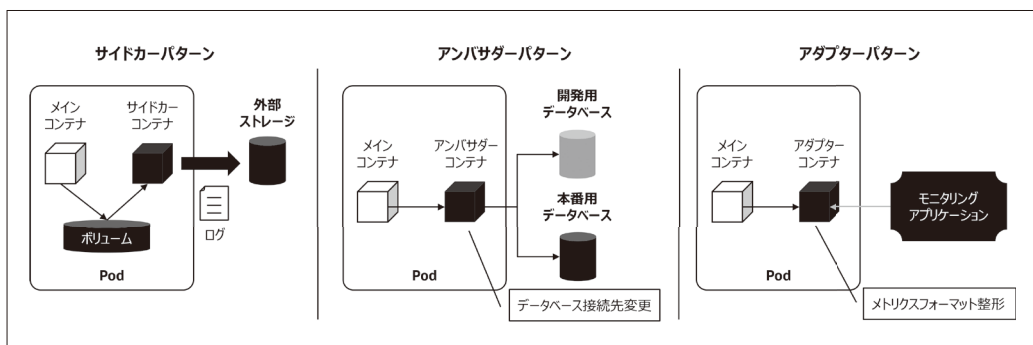


図-06 各パターンの主な用途例

マイクロサービスアーキテクチャでは、各サービスを効率的に分割および管理するために、Pod内のコンテナを役割ごとに分けるサイドカーパターンや、外部システムとの連携を容易にするアンバサダーパターンなどのデザインパターンを活用することで、柔軟性と拡張性の高いアプリケーション構成を実現できます。

>> ReplicaSet & Deployment

ReplicaSetは、Podのレプリカ（複製）を管理し、マニフェストで指定された数のPodを常に稼働させる役割を持つKubernetesのリソースです。

DeploymentはReplicaSetをさらに高度に管理するための上位リソースで、Podのスケールングやローリングアップデート、バージョン管理など、アプリケーションのライフサイクル全体を効率化します。

Deploymentを使用すると、ReplicaSetの作成や管理が自動化され、個別に操作する必要がなくなるため、アプリケーション運用が柔軟になります。これにより、スケールングやローリングアップデート、ロールバックといった高度な管理を一貫した方法で実行できます。

ReplicaSetとDeploymentの特徴を表-05に整理します。

特徴	概要
ReplicaSet	指定されたPodのレプリカ数と状態を維持し、Podが停止した場合には自動で再作成する (図-07)。
Deployment	ReplicaSetを管理し、Podのレプリカ数と状態を保証する。ローリングアップデートでダウンタイムなしの更新を実現し、ロールバックで過去の状態に戻すことも可能 (図-08)。

表-05 ReplicaSetとDeploymentの特徴

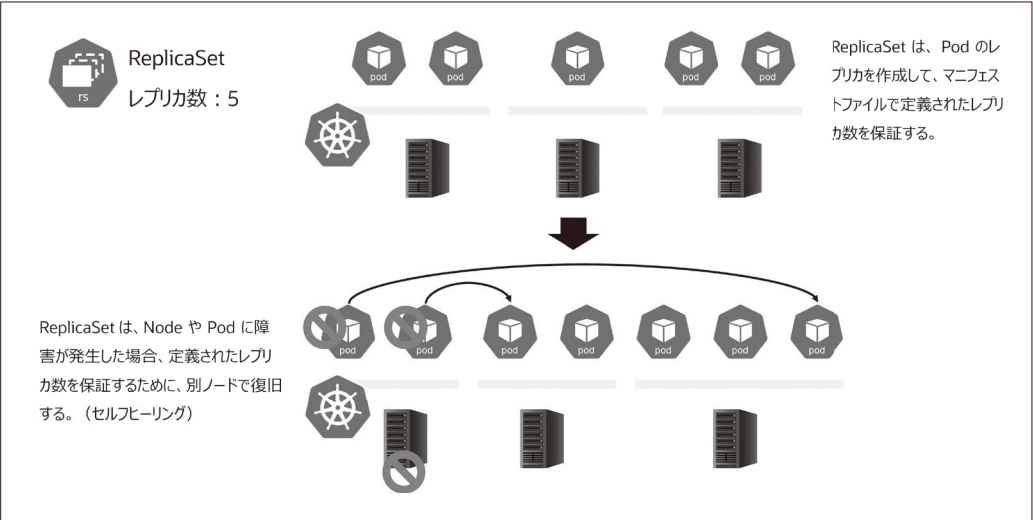


図-07 ReplicaSetセルフヒーリング

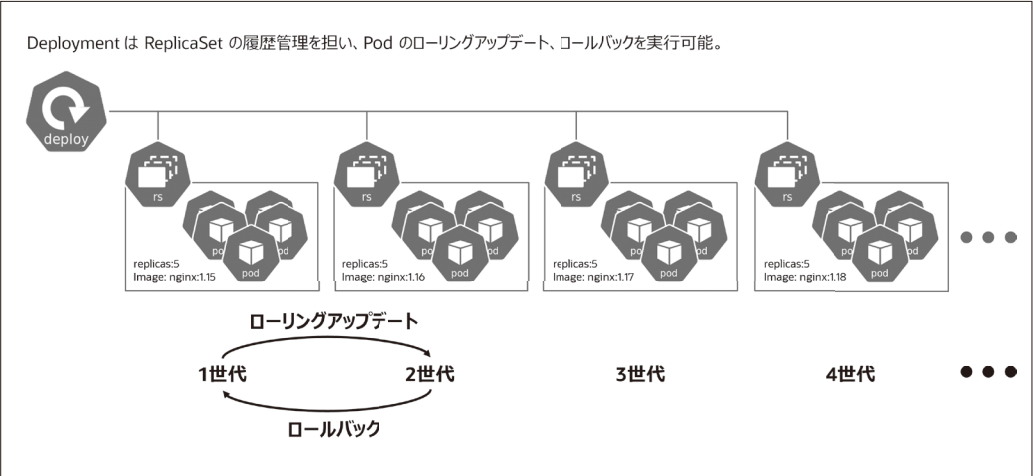


図-08 Deployment ローリングアップデート/ロールバック

Deploymentには、「Recreate」と「RollingUpdate」という2種類のローリングアップデートを実現する機能が実装されています。マニフェストに定義することで各機能を有効化できます。各機能の概要を表-06に整理します。

機能	概要
Recreate	既存のPodを全て削除してから新しいPodを作成する更新方法。一時的にアプリケーションが停止する可能性があるが、シンプルな更新に適している。
RollingUpdate	古いPodを徐々に停止しながら新しいPodを順次作成する更新方法。ダウンタイムなしで更新を実現する。Deploymentのデフォルト設定。

表-06 Deploymentの更新方法

ここまで、Pod、ReplicaSet、Deploymentを説明しました。それぞれの関係は、ReplicaSetがPodを管理し、そのReplicaSetをDeploymentが管理します(図-09)。

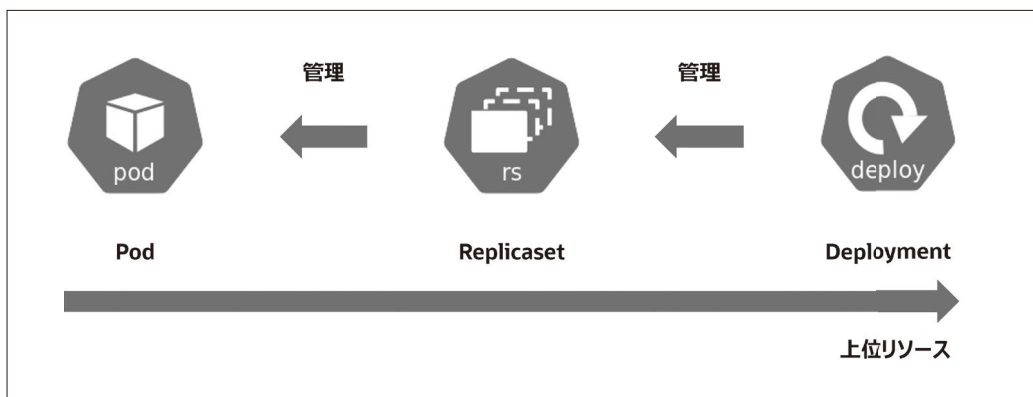


図-09 Pod、ReplicaSet、Deploymentの関係性

>> Service

Serviceは、Kubernetesクラスタ内外からPodへの安定したアクセスを提供します。Podは障害時やスケーリングによって再作成されるたびに新しいIPアドレスが割り当てられるため、直接通信には適していません。Serviceは、これら動的に変化するPodのIPアドレスを意識せずに通信できる仕組みを提供します。

Serviceにはいくつかのタイプがあり、それぞれ異なる用途や通信範囲を提供します。Serviceタイプを適切に選択することで、クラスタ内外のPodへの通信を効率的に管理できます。代表的なServiceタイプを表-07に整理します。

タイプ	概要	主な用途
ClusterIP	デフォルトのタイプで、クラスタ内からの通信のみを許可。仮想IPアドレスを割り当てる。	クラスタ内での内部通信（例：フロントエンドとバックエンド間の通信）。
NodePort	クラスタ内の各Nodeに固定のポートを割り当て、外部からNode IPとポートを使用してアクセス可能にする。	クラスタ外部からのシンプルなアクセス（例：ローカル開発環境でのテスト）。
LoadBalancer	クラウドサービスプロバイダのロードバランサーを利用して外部アクセスを提供。	クラウド環境での外部公開（例：Webアプリケーションの外部ユーザへの提供）。
ExternalName	外部DNS名を直接参照するタイプで、ClusterIPを持たない。	外部リソースの参照（例：サードパーティサービスや外部データベース）。
Headless Service	ClusterIPを持たず、DNSを通じて直接PodのIPアドレスを返す。	特定のPodに直接アクセスが必要な場合（例：データベースシャーディングやステートフル通信）。

表-07 Serviceのタイプ

Serviceタイプを選択する際には、ClusterIPはデフォルトのタイプであり、クラスタ内での通信をサポートする基本的な設定として利用されます。

NodePortやLoadBalancerは外部アクセスを提供するためのタイプで、NodePortは簡易的な外部アクセスに、LoadBalancerはクラウド環境での本番運用に適しています。

ExternalNameは外部リソースやサービスを参照する際に利用され、DNS名を直接使用してアクセスを可能にします。

一方、Headless Serviceは通常のServiceの抽象化を行わず、DNSを通じて特定のPodのIPアドレスに直接アクセスする必要がある特殊なケースで利用されます。

これらのタイプを理解することで、アプリケーションの要件や通信の特性に応じた最適なServiceの設定が可能となります。

主に利用される、ClusterIP (図-10)、LoadBalancer (図-11) について図解します。

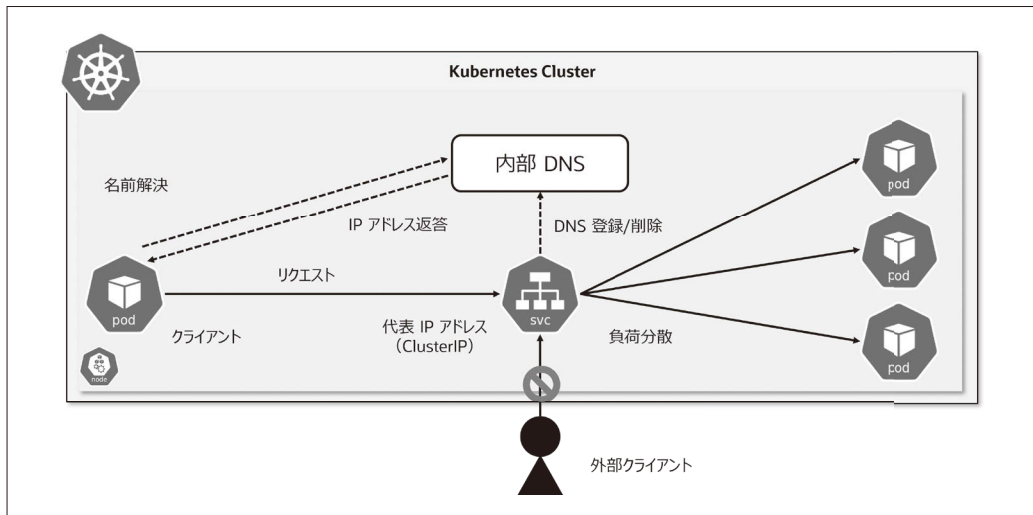


図-10 ClusterIP

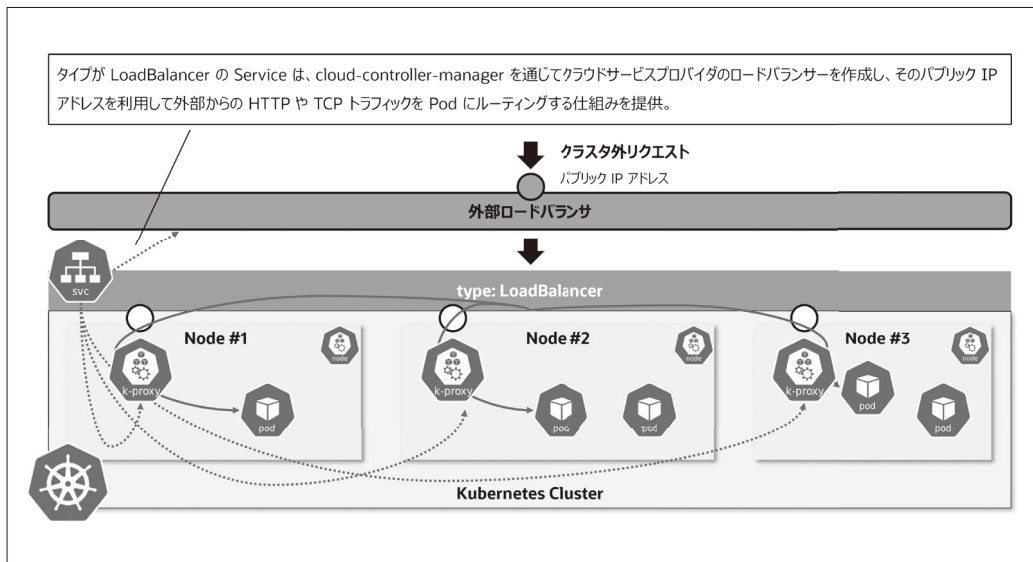


図-11 LoadBalancer

>> Ingress

Ingressは、外部からのHTTPやHTTPS (L7) リクエストをクラスタ内のServiceにルーティングします。Serviceがトランスポート層 (L4) の通信を管理し、IPアドレスやポートを基に

Podにトラフィックを送るのに対し、Ingressはアプリケーション層（L7）で動作し、ドメイン名やパスに基づく柔軟なリクエスト振り分けを実現します（図-12）。これにより、1つのIPアドレスで複数のアプリケーションを公開できます。

Ingressの動作には、リクエストを適切に処理するIngressコントローラ^{※2}（例：NGINX Ingress Controller^{※3}やTraefik^{※4}）が必要で、これと連携することで外部アクセスの効率的な管理と高度なルーティング機能を提供します。

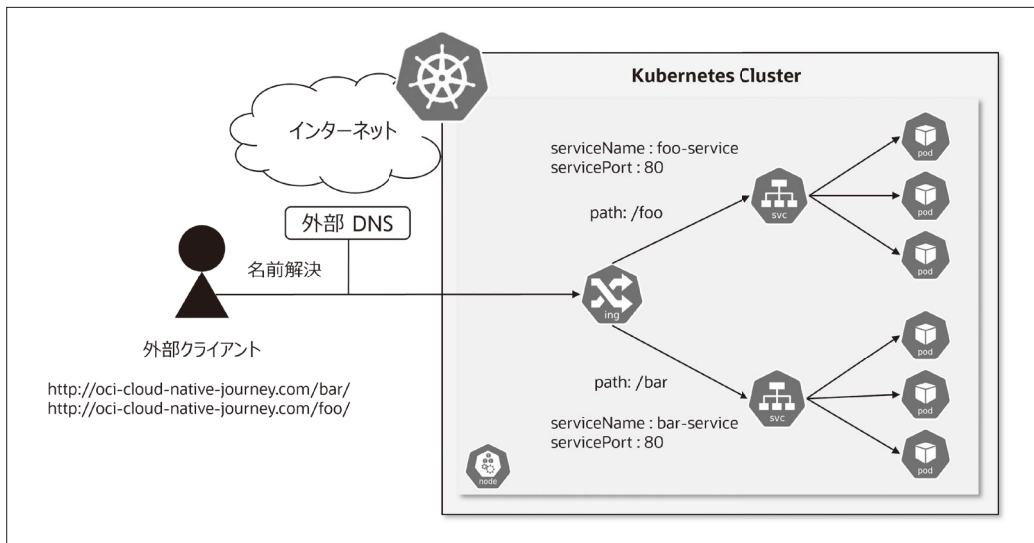


図-12 Ingressによるパスルーティング

補足として、2023年にGA（一般提供）となったGateway API^{※5}についても触れておきます。

Gateway APIは、Ingressを補完または置き換える形で設計された新しい標準です。Gateway APIは、HTTPやHTTPSに加えてTCPやUDPなど複数のプロトコルをサポートし、ネットワークトラフィックのルーティングをより柔軟かつ詳細に定義できます。

また、リソースをGateway、Routeなどに分割し、それぞれを独立して管理する仕組みを提供します。この分割により、複雑なネットワーク設定や複数チームによる運用管理が効率化さ

※2 Ingressコントローラは、Ingressリソースで定義されたルールに基づき、HTTPやHTTPSリクエストを適切なServiceにルーティングする役割を持つコンポーネント。

※3 NGINX Ingress Controller 公式GitHub： <https://github.com/kubernetes/ingress-nginx>

※4 traefik 公式ウェブサイト： <https://traefik.io/traefik/>

※5 Gateway API 公式ウェブサイト： <https://kubernetes.io/ja/docs/concepts/services-networking/gateway/>

れます。

IngressがL7通信の基本的なルーティングを提供するのに対し、Gateway APIは複数プロトコルや細かいトラフィック制御を必要とするユースケースに対応します。

>> PersistentVolume & PersistentVolumeClaim

PersistentVolume (PV) と PersistentVolumeClaim (PVC) は、Kubernetesにおけるストレージ管理を抽象化する仕組みです。

PVとPVCを説明する前に、Kubernetesにおけるデータの扱い方について一時性 (Temporary) と永続性 (Persistence) の観点から整理します。

一時性は、Kubernetesのクラスタ上でPodが削除または再デプロイされると同時にデータが消滅することを意味します (図-13)。

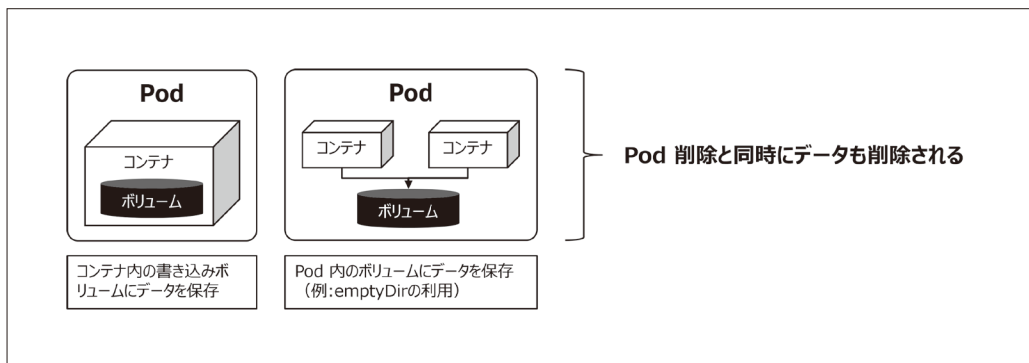


図-13 データの一時性

一方、永続性は、Kubernetesクラスタ上でPodが削除または再デプロイされてもデータが消滅しないことを意味します (図-14)。

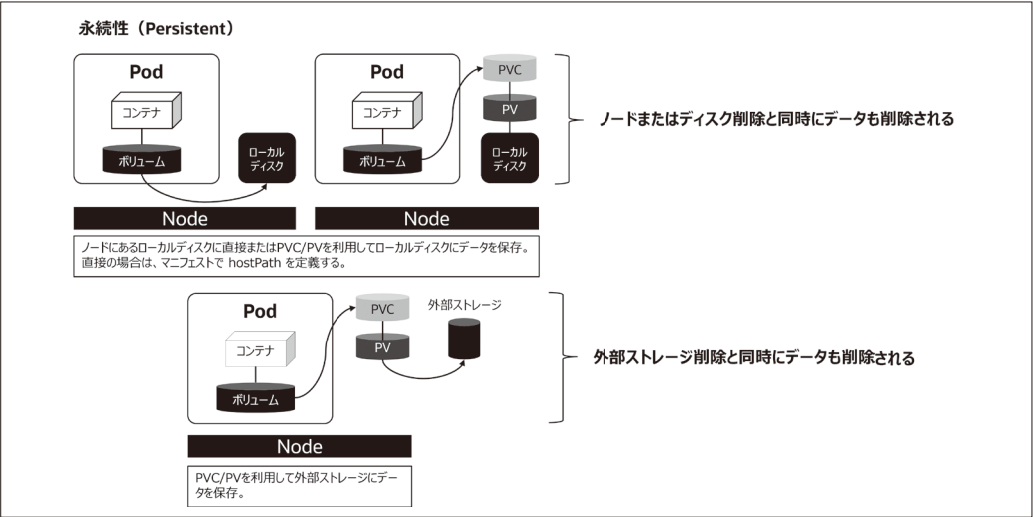


図 -14 データの永続性

PersistentVolume

PVは、Kubernetesでクラスタ内外のストレージを抽象化し、永続性のあるデータを保存するリソースです。ストレージバックエンドとしてクラウドストレージ（例：OCIストレージサービス）やローカルディスクを利用できます。PVは事前に管理者によってプロビジョニングされるか、動的にプロビジョニングされることで、PVCに応じたストレージを提供します。PVにおいて、永続化領域としてマニフェストで定義できる主な項目を表-08に整理します。

マニフェスト項目	概要
ラベル (metadata.labels)	PVにメタデータを付与するためのキーと値のペア。PVCによる選択に利用されることもある。
容量 (spec.capacity.storage)	PVが提供するストレージ容量を指定 (例：5Gi)。
アクセスモード (spec.accessModes)	ストレージへのアクセス方法を指定。以下の3種類が利用可能。： • ReadWriteOnce (単一Nodeから読み書き可能) • ReadOnlyMany (複数Nodeから読み取り可能) • ReadWriteMany (複数Nodeから読み書き可能) (図-15)
リクレームポリシー (spec.persistentVolumeReclaimPolicy)	PVがPVCから解放された後の動作を指定。(図-16、図-17)： • Retain (保持) • Delete (削除)
マウントオプション (spec.mountOptions)	ストレージのマウント時に使用する追加オプション。
ストレージクラス (spec.storageClassName)	PVを特定のストレージクラスに関連付けるための指定。

表 -08 永続化領域としてマニフェストで定義できる主な項目

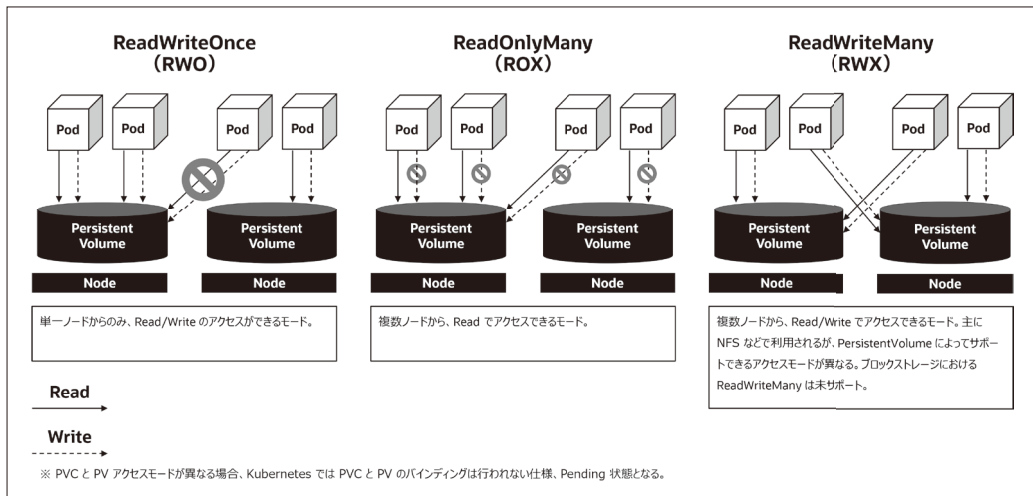


図-15 アクセスモードについて

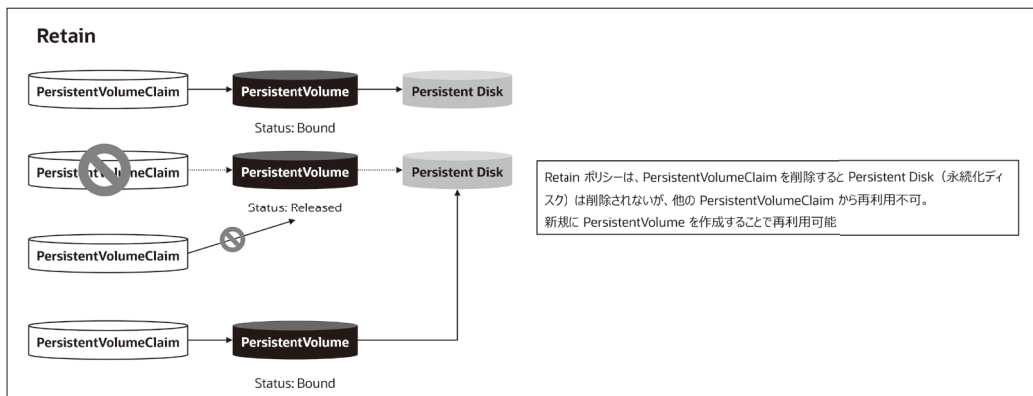


図-16 Reclaim Policy Retain

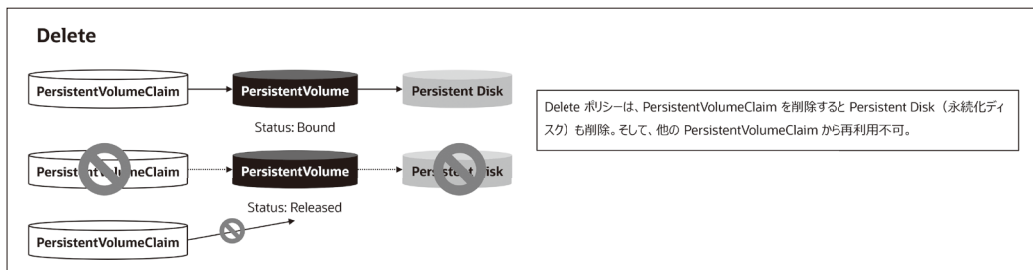


図-17 Reclaim Policy Delete

PersistentVolumeClaim

PVCは、開発者やアプリケーションが必要なストレージの容量やアクセスモードを指定して、Kubernetesクラスタ内で利用可能なPVを要求するためのリソースです。PVCはストレージのリクエストを抽象化し、Podがストレージを利用できるようにする仕組みを提供します。PVCは以下のステップで動作します。

- 1. 開発者がPVCを作成し、必要なストレージの条件（容量やアクセスモード）を指定
- 2. Kubernetesが指定された条件に一致するPVを探し、PVCにバインド
- 3. バインドされたPVは、Podによって利用可能

PVCにおいて、永続化要求としてマニフェストで定義できる主な項目を表-09に整理します。

マニフェスト項目	概要
ラベル (metadata.labels)	PVCにメタデータを付与するためのキーと値のペア。PVのラベルと一致する場合に選択されることがある。
容量 (spec.resources.requests.storage)	要求するストレージ容量を指定 (例：5Gi)。
アクセスモード (spec.accessModes)	ストレージへのアクセス方法を指定。以下の3種類が利用可能。 • ReadWriteOnce (単一Nodeから読み書き可能) • ReadOnlyMany (複数Nodeから読み取り可能) • ReadWriteMany (複数Nodeから読み書き可能) (図-15)
ストレージクラス (spec.storageClassName)	PVを特定のストレージクラスに関連付けるための指定。

表 -09 永続化要求としてマニフェストで定義できる主な項目

PVCはStorageClassを利用することで、動的プロビジョニングに対応しています。StorageClassをマニフェストで定義すると、必要なPVを自動的に作成できます。動的プロビジョニングでは、プロビジョナーと呼ばれるモジュールがPVCの作成に応じて自動的にPVを生成し、ストレージのプロビジョニングを効率化します (図-18)。

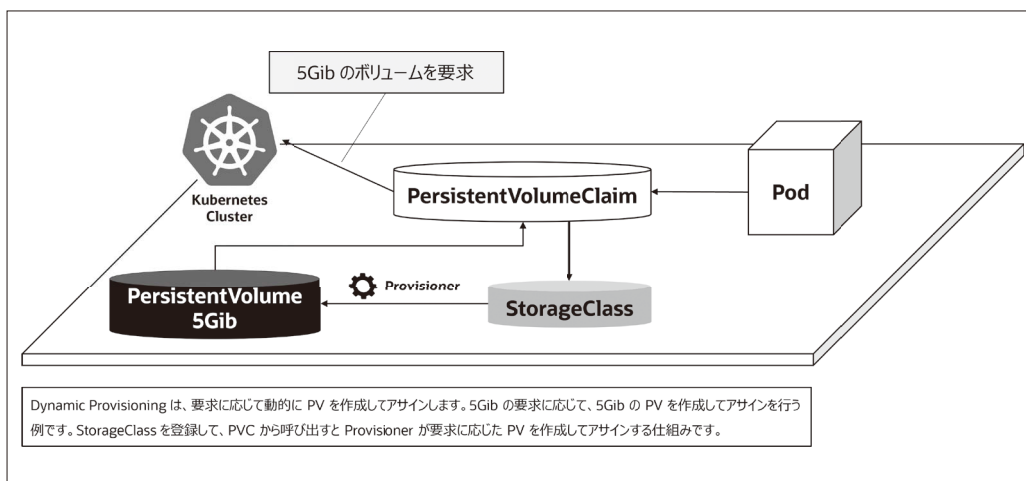


図-18 StorageClassによる動的プロビジョニング

OKEでは、ブロック・ボリューム・サービスとファイル・ストレージ・サービスに対して、StorageClassによる動的プロビジョニングに対応しています。詳細については、公式のドキュメント^{※6}を参照してください。

>> ConfigMap & Secret

ConfigMap

ConfigMapは、Kubernetes クラスタ内でアプリケーションの設定情報を管理します。アプリケーションのコンテナイメージとは分離された形で設定情報を保持することで、同じイメージを異なる環境（例：開発、ステージング、本番）で効率的に再利用できます（図-19）。

※6 公式ドキュメント：「CSIボリューム・プラグインを使用したブロック・ボリュームでのPVCの作成」https://docs.oracle.com/ja-jp/iaas/Content/ContEng/Tasks/contengcreatingpersistentvolumeclaim_topic-Provisioning_PVCs_on_BV.htm
 公式ドキュメント：「ファイル・ストレージ・サービスでのPVCのプロビジョニング」https://docs.oracle.com/ja-jp/iaas/Content/ContEng/Tasks/contengcreatingpersistentvolumeclaim_Provisioning_PVCs_on_FSS.htm

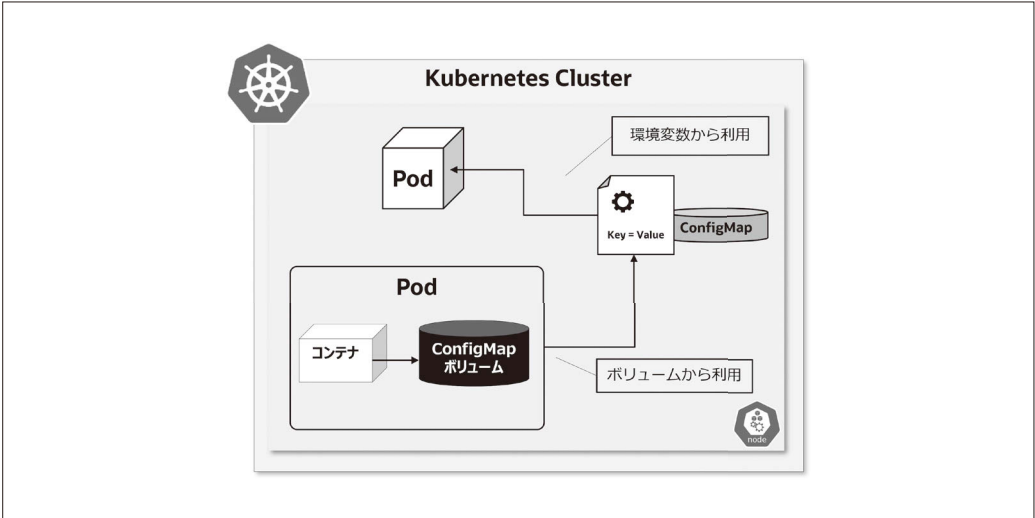


図 -19 ConfigMap概要図

主な特徴を表-10に整理します。

特徴	概要
設定の分離	コンテナイメージに設定情報を埋め込む必要がなく、設定変更を容易に行える。コンテナ数が多い場合により有効。
柔軟な設定形式	キーと値のペア、JSON/YAML形式のファイル、httpd.confのような設定ファイルを保持可能。
複数の利用方法	環境変数、コマンドライン引数、ボリュームマウントでコンテナに設定を渡すことが可能。
非機密性	平文で設定情報を管理するため、機密情報には不適。機密情報はSecretリソースで管理することを推奨。

表 -10 ConfigMapの特徴

Secret

Secretは、Kubernetes クラスタ内でパスワードやトークンといった機密情報を安全に管理します。ConfigMapと似ていますが、Secretは機密性を重視して設計されており、設定情報を暗号化された形式で保持します。これにより、機密データをアプリケーションに渡す際のセキュリティが強化されます (図-20)。

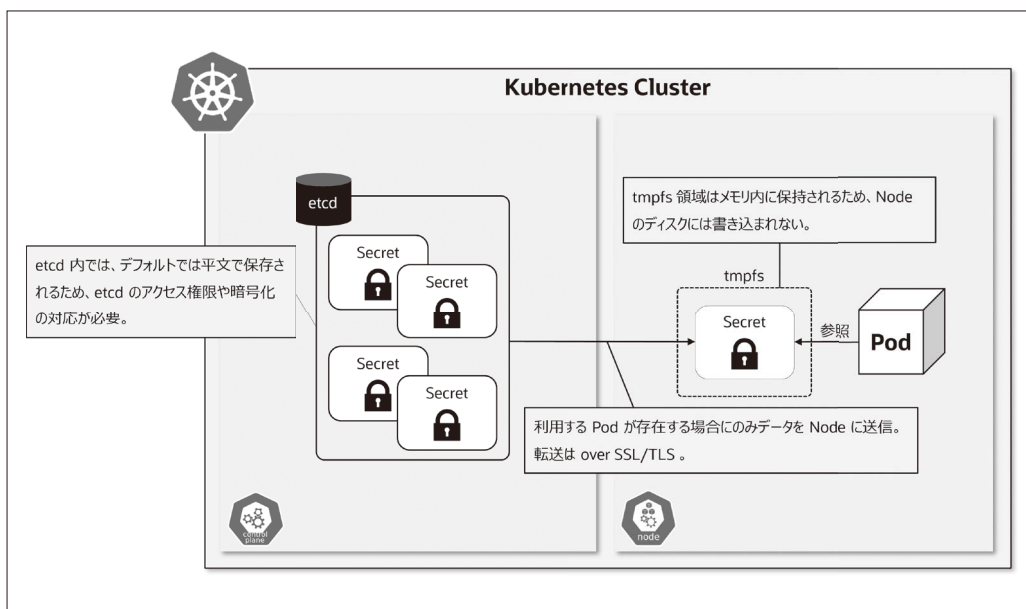


図-20 Secret概要図

主な特徴を表-11 に整理します。

特徴	概要
機密情報の管理	ID/Password、API キー、TLS 証明書などの機密データを安全に保存。
暗号化対応	Secret は etcd を暗号化することで、Kubernetes クラスタ内で暗号化して保存が可能。
複数の利用方法	環境変数、ボリュームマウント、imagePullSecrets として機密情報を Pod に渡すことが可能。
制限付きアクセス	RBAC (Role-Based Access Control) を利用して Secret へのアクセスを細かく制御可能。

表-11 Secretの特徴

Secret にデータを格納する際に Base64 にエンコードする必要があります。エンコードされているだけで暗号化されていません。Secret が定義されたマニフェストを外部の Git リポジトリで管理する場合にセキュリティリスクを伴います。OCI では、Vault サービス^{※7}を利用することで回避できます。オープンソースソフトウェアでは、Kubesecc^{※8}、External Secret^{※9}、

※7 OCI Vault 公式ドキュメント：「OCI Vault」<https://docs.oracle.com/ja-jp/iaas/Content/KeyManagement/home.htm>

※8 Kubesecc 公式 GitHub：<https://github.com/shyiko/kubesecc>

※9 External Secret 公式 GitHub：<https://github.com/external-secrets/external-secrets>

Sealed Secrets^{※10}があります。各クラウドサービスプロバイダが提供する専用サービスもあります。OCIでは、OCI Secrets Store CSI Driver Provider^{※11}があります。

>> Namespace

Namespaceは、Kubernetesクラスタ内でリソース (Pod、Service、ConfigMap など) を論理的に分離する仕組みです。大規模なクラスタやマルチテナント環境で、リソースの整理やアクセス制御を効率的に行うために使用されます。Namespaceを利用することで、異なるプロジェクトやチームが1つのクラスタを共有しつつ、リソースの干渉を防ぐことができます (図-21)。

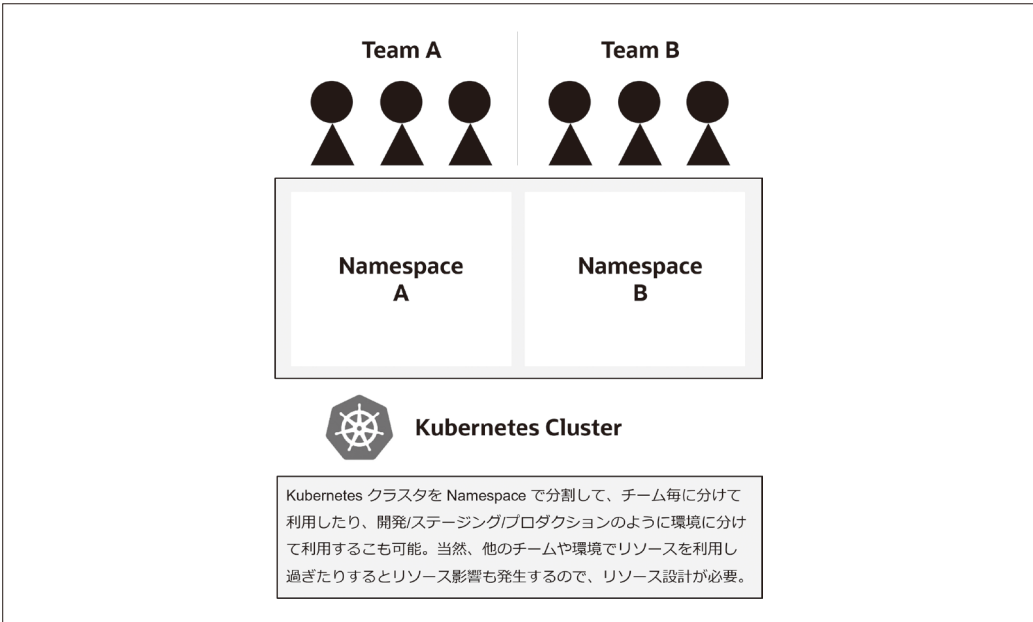


図-21 Namespace概要図

主な特徴を表-12に整理します。

※10 Sealed Secrets 公式GitHub : <https://github.com/bitnami-labs/sealed-secrets>
※11 OCI Secrets Store CSI Driver Provider 公式GitHub : <https://github.com/oracle/oci-secrets-store-csi-driver-provider>

特徴	概要
リソースの分離	各Namespace内のリソースは独立しており、他のNamespaceのリソースに影響を与えない。
アクセス制御	RBACと組み合わせてNamespace単位でユーザーやアプリケーションのアクセス権を細かく制御可能。
リソースの整理	同じ名前のリソースを異なるNamespace内で作成可能、名前の競合を避けられる。
リソースの割り当て制限	ResourceQuotaを使用してNamespaceごとにCPUやメモリの使用量を制限可能。
デフォルトのNamespace	明示的に指定しない場合、default Namespaceが使用される。
使用例	- 異なるチームやプロジェクトごとにリソースを分離 - ステージング、開発、本番環境を分けて管理 - マルチテナント環境でテナントごとに分離

表-12 Namespaceの特徴

>> Label & Label Selector

ここからは、Kubernetesを利用する上で必要となるLabelとLabel Selectorについて説明します。

Label

Labelは、Kubernetesリソースに付与するキーと値のペアで、リソースの分類や識別に使用されるメタデータです。任意のキーと値を指定でき、リソースの検索や管理を効率化するために活用されます。

主な特徴を表-13に整理します。

特徴	概要
リソースの分類と識別	キーと値のペアによるメタデータの付与による分類および識別。
柔軟な定義	任意のキーと値による自由なラベル設定。
管理の効率化	リソースの検索、選択、管理の簡略化。

表-13 Labelの特徴

Label Selector

Label Selectorは、ラベルの条件に基づいてKubernetesリソースを選択する仕組みです。特定のラベルを持つリソースのみを対象に操作、監視できるため、大規模な環境でも効率的な管理が可能です。

主な特徴を表-14に整理します。

特徴	概要
リソースの動的選択	ラベル条件に基づく動的なリソースの選択。
条件指定の柔軟性	等値ベース (例: key=value) や集合ベース (例: key in (value1, value2)) で条件を指定可能。
幅広い対応リソース	Deployment、Service、ReplicaSetなど、多くのKubernetesリソースで使用される。

表-14 Label Selectorの特徴

 Helm の基礎知識

Helm について

Helm^{※12}は、Kubernetes 向けのパッケージマネージャーです。複数の Kubernetes リソースを一括で管理およびデプロイするためのツールで、アプリケーションのインストール、更新、削除を簡略化します。Helm を使うことで、複雑なアプリケーションの構成をテンプレート化し、再利用性や保守性を高めることができます。

パッケージは Chart と呼ばれ、Helm ではこれを集約して管理するリポジトリが存在します。Chart には Kubernetes リソースのテンプレートがまとめられており、複数のリソースを効率的にデプロイできます。Helm は、Linux における dnf や apt に相当し、Chart は rpm や deb に相当すると考えると理解しやすいでしょう。つまり、Helm を使うことで、Chart という単位でパッケージ化された Kubernetes アプリケーションをリポジトリから取得し、簡単にインストール、アップデート、削除ができるようになります。リポジトリには公式チャートだけでなく、コミュニティやカスタムチャートを公開することもでき、柔軟に運用できます。主な特徴を表-15に整理します。

※12 Helm 公式ウェブサイト：<https://helm.sh/ja/>

特徴	概要
簡単なアプリケーションデプロイ	事前に定義されたテンプレート (Chart) を使用し、複数の Kubernetes リソースを一括でデプロイできる。
柔軟なカスタマイズ	Chart はテンプレート化されており、環境に応じた設定変更やカスタマイズが容易。
リリース管理	アプリケーションをリリースとして管理し、履歴を保持。ロールバック機能により安全にアップデートできる。
Helm リポジトリ	Chart を保存および共有するリポジトリを提供。公式リポジトリやカスタムリポジトリから Chart を取得して利用できる。
オープンソースで広く利用	公式やコミュニティ提供の豊富な Chart を利用できる。多くのアプリケーションを Kubernetes クラスタに簡単に導入できる。

表-15 Helmの特徴

Chart と values ファイルの関係性

Chart は、Kubernetes リソース (Pod、Service、ConfigMap など) のテンプレートをまとめた Helm パッケージです。このテンプレートには変数が含まれており、実際に使用する際に具体的な値を設定する必要があります。その設定で使用するのが、values ファイルです。

values ファイルは、Chart のテンプレートに渡す具体的な値を定義する設定ファイルで、YAML 形式で記述します。これにより、Chart を様々な環境 (例: 開発、ステージング、本番) や条件に合わせて簡単にカスタマイズできます (図-22)。

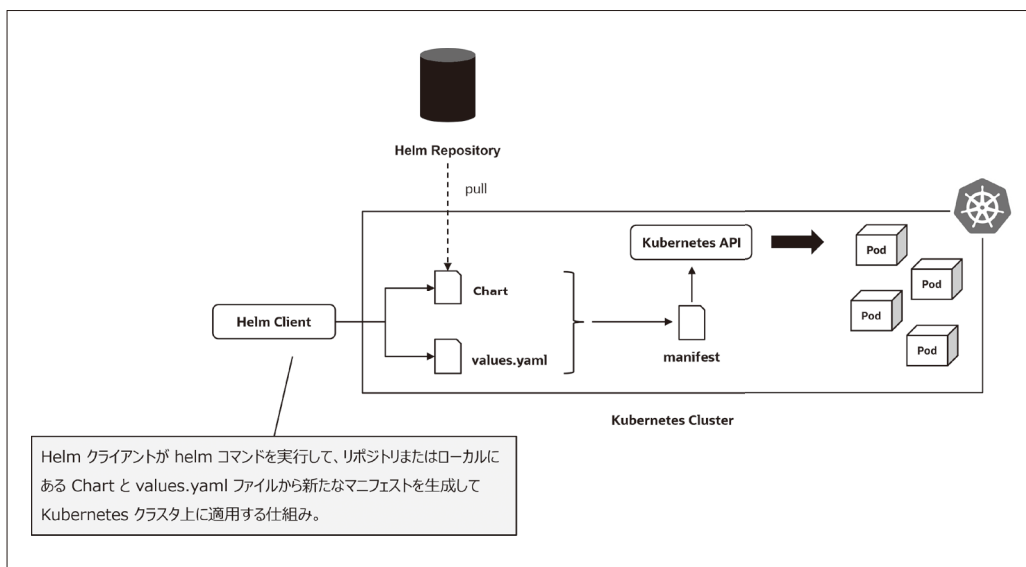


図-22 Helm適用の概要図

Helmを利用してKubernetesクラスタ上にNginxのPodを適用するまでの流れを例に、Chartとvaluesファイルの関係性を確認します。

Helmを利用する場合、Helmクライアントが必要となります。OCI Cloud Shellでは、あらかじめHelmクライアントが利用できるため、OCI Cloud Shell起動後にHelmコマンドを実行できます。

≫ 1. Chart の作成

以下のコマンドで、新しいHelm Chartを作成します。

```
$ helm create my-pod-chart
```

このコマンドにより、my-pod-chartという名前のディレクトリが作成され、その中にデフォルトのChartテンプレート構造が生成されます。

≫ 2. Chart テンプレートの編集

デフォルトのtemplatesディレクトリ内のファイルを削除した後に、PodのChartテンプレートを作成します。

Chartテンプレートファイル：「my-pod-chart/templates/pod.yaml」

```
$ rm -rf my-pod-chart/templates/*
```

```
$ vim my-pod-chart/templates/pod.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: {{ .Values.appName }}
  labels:
    app: {{ .Values.appName }}
spec:
  containers:
    - name: {{ .Values.appName }}
      image: {{ .Values.image }}
```

```
ports:
- containerPort: {{ .Values.containerPort }}
env:
- name: LOG_LEVEL
  value: {{ .Values.logLevel }}
```

- Chartテンプレートの役割：

Chartテンプレートは、Kubernetesリソースの定義をテンプレート化したものです。「{{ .Values.XXX }}

- サンプルChartテンプレートの内容：

Podの構成を定義しており、名前やイメージ、ポート、環境変数などがvaluesファイルから動的に設定されます。

>> 3. values ファイルの編集

デフォルトで作成される values.yaml を以下の通りに編集します。

values ファイル：「my-pod-chart/values.yaml」

```
$ vim my-pod-chart/values.yaml
```

```
appName: my-app
image: nginx:latest
containerPort: 80
logLevel: debug
```

- values ファイルの役割：

values ファイルは、テンプレートで使用する値を定義するファイルです。環境ごとに異なる値を指定することで、テンプレートを再利用できます。更新する場合も、values ファイルの値を変更するだけで反映できるため、運用効率が高まります。

- サンプル values ファイルの内容：

appName：Podやコンテナの名前に使用される値を定義
 image：使用するコンテナイメージ（例：nginx:latest）を指定
 containerPort：コンテナが公開するポート番号を指定

logLevel：アプリケーションのログ出力レベルを指定

>> 4. Helm Chart のインストール

以下のコマンドでHelm Chartをインストールします。

```
$ helm install my-pod-release ./my-pod-chart
```

```
NAME: my-pod-release
LAST DEPLOYED: Mon Jan 6 12:33:32 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

- my-pod-release :
このインストールのリリース名 (任意の名前に変更可能)
- ./my-pod-chart :
Chartディレクトリのパス

>> 5. インストールの確認

インストールされたりソースを確認します。

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
my-app	1/1	Running	0	11s

my-app という名前のPodがデプロイされていることを確認できます。

>> 6. 更新と削除

- 設定の変更 :
values ファイルを編集後、以下のコマンドで変更を反映できます。更新の度に、表示結果のREVISIONの値が加算されます。

```
$ helm upgrade my-pod-release ./my-pod-chart
```

```
Release "my-pod-release" has been upgraded. Happy Helming!
NAME: my-pod-release
LAST DEPLOYED: Mon Jan 6 12:37:06 2025
NAMESPACE: default
STATUS: deployed
REVISION: 2
TEST SUITE: None
```

- リリースの削除：

Helm でインストールしたリソースを削除するには以下のコマンドを使用します。

```
$ helm uninstall my-pod-release
```

```
release "my-pod-release" uninstalled
```

このように、Helm を使うと Kubernetes リソースを効率的に管理できます。Chart はリソースのテンプレートを提供し、values ファイルで具体的な値を埋め込むことで、異なる環境や条件に合わせて再利用可能です。この仕組みにより、個別のマニフェストを手作業で作成する負担を減らし、リソースの一貫性を保ちながら柔軟で効率的な運用を実現します。

Terraform の基礎知識

Terraform とは？

Terraform は、HashiCorp 社が開発したインフラストラクチャ自動化ツールで、主にクラウド環境でのインフラの構築や構成管理を効率化するために使われます。これは「Infrastructure as Code (IaC)」のコンセプトを採用しており、コードを用いてインフラの設計と運用を行います。

Terraform の特徴は、インフラの「あるべき状態」を宣言的に記述し、その状態を実現するために必要な API リクエストを Terraform が自動的に計算、実行する点です。これにより、

手続き的なスクリプトを書く必要がなく、コードそのものがインフラの最終形を示すドキュメントとして機能します。また、現在のリソースの状態を明示的に意識する必要がなくなるため、開発者や運用者の負担を軽減します。

Terraformでは、宣言的な構成管理のために「HashiCorp Configuration Language (HCL)」を使用します。このDSL（ドメイン特化言語）※¹³はJSON互換で、柔軟かつ可読性に優れた構文を提供します。Terraformは、設定された「あるべき状態」と「現状」を比較し、必要な変更を適用するために「Stateファイル」を使用します。このStateファイルに基づき、冪等性（同じコードを複数回実行しても結果が変わらない性質）が確保されます。これにより、安全かつ確実にインフラの更新が行えます。

Terraform のアーキテクチャ

Terraformは、プロバイダーが提供するAPIを利用してインフラを構築します。この仕組みは、3つの主要コンポーネントで構成されています（図-23）。

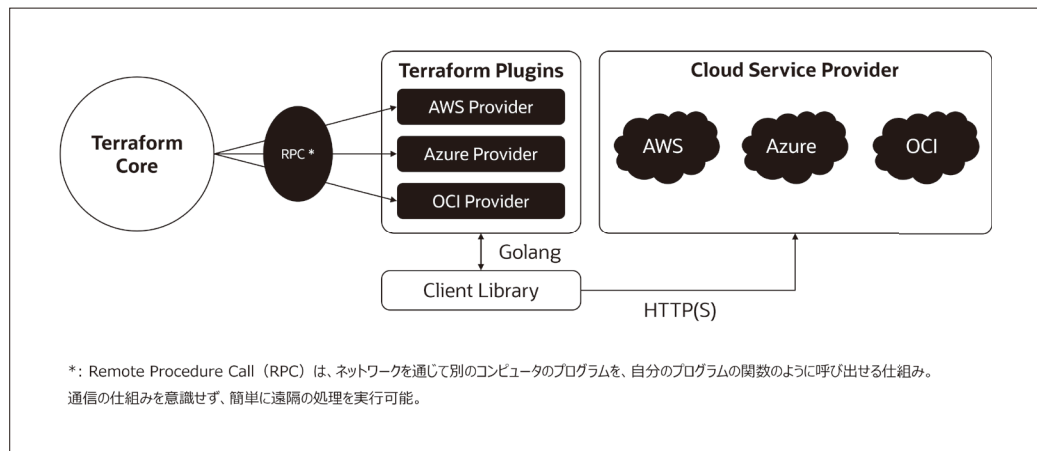


図-23 Terraformアーキテクチャ概要

各コンポーネント概要を表-16に整理します。

※¹³ Domain Specific Language の略で、特定の分野や用途に特化して設計されたプログラミング言語や構文のこと。

コンポーネント	概要
Terraform Core	Terraformの基本機能を提供し、CLI操作、依存関係グラフの作成、Stateファイルの管理を行う。HashiCorp社が所有し、GitHubのリポジトリで管理されている。
Terraform Plugins	各プロバイダーのAPIを操作するためのモジュールで、Terraform Coreがこれを利用してインフラを構築。各プラグインのコードはそれぞれ個別のGitHubリポジトリで管理されている（例：oracle/terraform-provider-oci）。
Terraform Providers	各プロバイダーのAPIに接続する具体的なプラグイン（例：terraform-provider-oci）。Official、Partner、Community含めて数多くのプロバイダー ^{*14} が登録されている。

表-16 Terraform コンポーネント概要

Terraform によるプロビジョニングの流れ

Terraform によるプロビジョニングの流れは以下の通りです。

1. 構成の記述：HCLを使ってインフラの「理想の状態」をコードに記述
2. 初期化（terraform init）：必要なプロバイダーやプラグインを準備
3. 計画の確認（terraform plan）：現在の状態と理想の状態を比較し、必要な変更を確認
4. 適用（terraform apply）：計画された変更を実行し、インフラを構築または更新
5. 状態の管理：tfstate ファイルに現在のインフラ状態を記録し、次回以降の変更に備える
6. 削除の実行（terraform destroy）：インフラを削除し、関連リソースを解放します。

※削除する場合

この流れを繰り返すことで、効率的にインフラを管理できます。全体のイメージを表したのが図-24です。

※14 Terraform Providers 公式ウェブサイト：<https://registry.terraform.io/browse/providers>

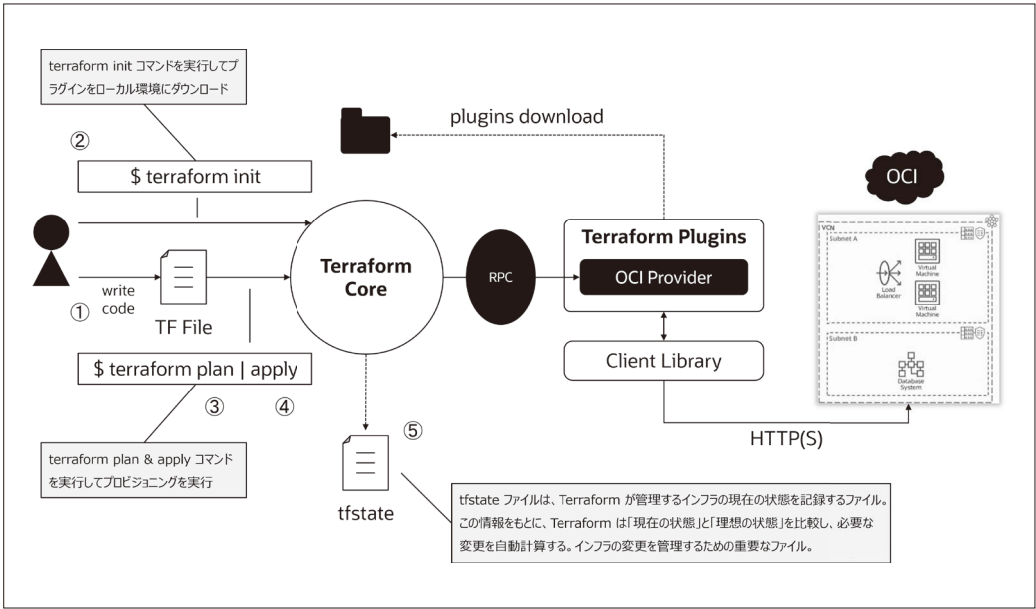


図-24 Terraformによるプロビジョニングの流れ

Terraform による OCI Object Storage Bucket の作成

OCI Cloud Shell には、Terraform クライアントがインストールされています。事前に認証設定を行うことで、terraform コマンドを利用したプロビジョニングを実行できます。

OCI Cloud Shell を利用して、実際に OCI Object Storage のバケットをプロビジョニングしてみます。

>> 1. OCI Cloud Shell の起動

OCI Console Dashboard の上部メニューにあるアイコンをクリックして、「Cloud Shell」を選択します。OCI Cloud Shell が起動します (図-25)。



図-25 OCI Cloud Shell の起動

>> 2. 公開鍵と秘密鍵の作成

OCI Cloud Shell起動後、認証に必要な公開鍵と秘密鍵を作成します。最初に専用のディレクトリを作成します。

```
$ mkdir $HOME/.oci
```

opensslコマンドを実行して秘密鍵を作成します。

```
$ openssl genrsa -out $HOME/.oci/terraform.pem 2048
```

作成した秘密鍵に権限を設定します。

```
$ chmod 600 $HOME/.oci/terraform.pem
```

公開鍵を作成します。

```
$ openssl rsa -pubout -in $HOME/.oci/terraform.pem -out $HOME/.oci/terraform_public.pem
```

公開鍵の内容を表示します。

```
$ cat $HOME/.oci/terraform_public.pem
```

※公開鍵の内容は省略します。後続の手順で必要となりますので、メモ帳またはテキストエディタにペーストして保存しておきます。

>> 3. APIキーの登録

catコマンドで表示した公開鍵をOCIのAPIキーとして登録します。

OCI Console Dashboardの上部メニューのプロファイルアイコンをクリックして、「ユーザー設定」を選択します(図-26)。

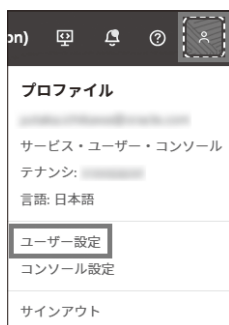


図 -26 APIキーの登録01

次に左下にあるリソースメニューの「APIキー」を選択します (図-27)。



図 -27 APIキーの登録02

「APIキーの追加」ボタンをクリックします (図-28)。

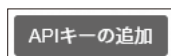


図 -28 APIキーの登録03

「公開キーの貼付け」を選択して、事前にcatコマンドで取得した公開鍵をコピーして、「公開キー」の箇所にペーストします。その後、「追加」ボタンをクリックします (図-29)。

APIキーの追加 [ヘルプ](#)

ノート: APIキーは、APIリクエストの署名に使用されるPEM形式のRSAキー・ペアです。ここでキー・ペアを生成して秘密キーをダウンロードできます。すでにキー・ペアを保持している場合は、かわりに公開キー・ファイルをアップロードするか、貼り付けることを選択できます。 [さらに学ぶ](#)

☐ APIキー・ペアの生成
 ☐ 公開キー・ファイルの選択
 ☒ 公開キーの貼付け

公開キー

追加

図-29 APIキーの登録04

「閉じる」ボタンをクリックします (図-30)。

構成ファイルのプレビュー [ヘルプ](#)

ノート: この構成ファイル・スニペットには、SDK、CLIまたはその他のOCI開発者ツールを使用するために必要なBasic認証情報が含まれます。テキスト・ボックスの内容を~/oci/configファイルに貼り付け、秘密キーへのファイル・パスを使用してkey_fileパラメータを更新します。構成プロファイルにすでにデフォルトのプロファイルがある場合は、追加のステップをいくつか実行する必要があります。 [さらに学ぶ](#)

APIキー・フィンガープリントの選択

構成ファイルのプレビュー 読取り専用

テキスト・ボックスの内容を~/oci/configファイルに貼り付けます。 [コピー](#)

閉じる

図-30 APIキーの登録05

リストに表示されるフィンガープリントは、後述のterraformのソースコード内に設定します (図-31)。

フィンガープリント	作成日
[Masked]	2025年1月8日(水) 8:18:39 UTC

1個のAPIキーを表示中

図-31 APIキーの登録06

>> 4. 設定情報の取得

以下の情報を取得しておきます。

- テナント OCID
- コンパートメント OCID
- ユーザー OCID
- フィンガープリント
- リージョン名
- API 署名用の秘密鍵のパス

>> 5. tf ファイルの作成

OCI Cloud Shell 上のホームディレクトリに専用のディレクトリを作成します。

```
$ mkdir tf-os
```

Cloud Service Provider (OCI) に API リクエストを送るために必要な設定ファイルを「terraform.tf」として作成します。

```
$ vim tf-os/terraform.tf
```

```
provider "oci" {  
  region = var.region  
}  
  
terraform {
```

```
required_providers {
  oci = {
    source = "oracle/oci"
    version = "< 6.0.0"
  }
}
```

事前取得した情報を変数として利用するため、「variables.tf」として作成します。

```
$ vim tf-os/variables.tf
```

```
# Provider
variable "region" {
  description = "OCI Region Identifier. (e.g. ap-tokyo-1, ...)"
  default     = "プロビジョニングするリージョンコード"
}

variable "tenancy_ocid" {
  description = "OCID of tenancy."
  default     = "プロビジョニングするテナントのOCID"
}

variable "user_ocid" {
  description = "OCID of user."
  default     = "プロビジョニングするテナントユーザーのOCID"
}

variable "private_key_path" {
  default     = "$HOME/.oci/terraform.pem" # 秘密鍵のパス
  description = "File path of private key."
}

variable "private_key" {
  default     = null
  description = "Content of private key."
}

variable "fingerprint" {
```

```
description = "Fingerprint."
default     = "APIキーのフィンガープリント"
}

variable "compartment_ocid" {
  description = "OCID of compartment."
  default     = "プロビジョニングするコンパートメントのOCID"
}

# Object Storage
variable "objectstorage_namespace" {
  description = "Namespace of Object Storage."
  default     = "xxxxx" # 任意の値
}

variable "objectstorage_bucket_name" {
  description = "Name of Object Storage Bucket."
  default     = "terraform" # 任意の値
}
```

「main.tf」に、オブジェクトストレージサービスのバケットをプロビジョニングする定義を書きます。

```
$ vim tf-os/main.tf
```

```
resource "oci_objectstorage_bucket" "objectstorage_bucket" {
  compartment_id = var.compartment_ocid
  name           = var.objectstorage_bucket_name
  namespace      = var.objectstorage_namespace
}
```

≫ 6. init、plan、apply、destroyの実行

専用のディレクトリに移動します。

```
$ cd tf-os
```

初期化を実行します。

```
$ terraform init
```

```
Initializing the backend...
```

```
Initializing provider plugins...
```

- Reusing previous version of oracle/oci from the dependency lock file
- Using previously-installed oracle/oci v5.47.0

```
Terraform has been successfully initialized!
```

You may now begin working with Terraform. Try running "terraform plan" to see any changes that are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.

計画を実行して、プロビジョニングされるリソースをプレビューします。

※出力の一部をマスクしています。

計画を実行します。

```
$ terraform plan
```

```
Initializing the backend...
```

```
Initializing provider plugins...
```

- Reusing previous version of oracle/oci from the dependency lock file
- Using previously-installed oracle/oci v5.47.0

```
Terraform has been successfully initialized!
```

You may now begin working with Terraform. Try running "terraform plan" to see any changes that are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.

```
xxxxxx@cloudshell:tf-os (xx-xxxxx-x)$ vim terraform.tf
```

```
xxxxxx@cloudshell:tf-os (xx-xxxxx-x)$ terraform plan
```

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:

- + create

Terraform will perform the following actions:

```
# oci_objectstorage_bucket.objectstorage_bucket will be created
```

```
+ resource "oci_objectstorage_bucket" "objectstorage_bucket" {
  + access_type           = "NoPublicAccess"
  + approximate_count     = (known after apply)
  + approximate_size      = (known after apply)
  + auto_tiering          = (known after apply)
  + bucket_id             = (known after apply)
  + compartment_id        = "ocidl.compartment.oc1.."
  + created_by            = (known after apply)
  + defined_tags          = (known after apply)
  + etag                  = (known after apply)
  + freeform_tags         = (known after apply)
  + id                    = (known after apply)
  + is_read_only          = (known after apply)
  + kms_key_id            = (known after apply)
  + name                  = "terraform"
  + namespace             = "xxxxx"
  + object_events_enabled = (known after apply)
  + object_lifecycle_policy_etag = (known after apply)
  + replication_enabled   = (known after apply)
  + storage_tier          = (known after apply)
  + time_created          = (known after apply)
  + versioning            = (known after apply)
}
```

Plan: 1 to add, 0 to change, 0 to destroy.

```

-----
-----
-----

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to
take exactly these actions if you run
"terraform apply" now.

```

この時、プロバイダー定義に必要な変数やリソース定義に必要な変数を入力する方法は、以下の通り複数存在します。

- variables.tf の default 属性にデフォルト値を記載する
- 環境変数 (TF_VAR_xxx) から渡す
- terraform.tfvars(json) ファイルから渡す
- *.auto.tfvars(json) ファイルから渡す
- CLI の実行時オプションとして渡す (-var="xxx=yyy", or -var-file="xxx.tfvars")
- 実行時に対話形式で指定する

初期化、計画が完了したので、適用します。

※出力の一部をマスクしています。

```
$ terraform apply
```

```

Terraform used the selected providers to generate the following execution plan. Resource
actions are indicated with the
following symbols:

```

```
+ create
```

```
Terraform will perform the following actions:
```

```

# oci_objectstorage_bucket.objectstorage_bucket will be created
+ resource "oci_objectstorage_bucket" "objectstorage_bucket" {
  + access_type           = "NoPublicAccess"
  + approximate_count     = (known after apply)
  + approximate_size      = (known after apply)

```

```
+ auto_tiering           = (known after apply)
+ bucket_id             = (known after apply)
+ compartment_id        = "ocid1.compartment.oc1.."
+ created_by            = (known after apply)
+ defined_tags           = (known after apply)
+ etag                  = (known after apply)
+ freeform_tags          = (known after apply)
+ id                    = (known after apply)
+ is_read_only           = (known after apply)
+ kms_key_id            = (known after apply)
+ name                  = "terraform"
+ namespace             = "xxxxx"
+ object_events_enabled  = (known after apply)
+ object_lifecycle_policy_etag = (known after apply)
+ replication_enabled    = (known after apply)
+ storage_tier           = (known after apply)
+ time_created           = (known after apply)
+ versioning            = (known after apply)
}
```

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?

Terraform will perform the actions described above.

Only 'yes' will be accepted to approve.

「yes」と入力して実行します。

Enter a value: yes

oci_objectstorage_bucket.objectstorage_bucket: Creating...

oci_objectstorage_bucket.objectstorage_bucket: Creation complete after 1s [id=n/orasejapan/b/terraform]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

リソースが作成されたことを確認します。

※出力の一部をマスクしています。

```
$ terraform show
```

```
# oci_objectstorage_bucket.objectstorage_bucket:
resource "oci_objectstorage_bucket" "objectstorage_bucket" {
  access_type           = "NoPublicAccess"
  approximate_count     = "0"
  approximate_size      = "0"
  auto_tiering          = "Disabled"
  bucket_id             = "ocid1.bucket.oc1.xx-xxxxx-xx."
  compartment_id        = "ocid1.compartment.oc1.."
  created_by            = "ocid1.user.oc1.."
  defined_tags          = {}
  etag                  = "5c39bd10-f705-48f6-a9ca-00ed480ac661"
  freeform_tags         = {}
  id                    = "n/xxxxx/b/terraform"
  is_read_only          = false
  name                  = "terraform"
  namespace             = "xxxxx"
  object_events_enabled = false
  replication_enabled   = false
  storage_tier          = "Standard"
  time_created          = "2025-01-08 10:21:50.068 +0000 UTC"
  versioning            = "Disabled"
}
```

terraform apply を実行したディレクトリには、「terraform.tfstate」というファイルが作成されています。「terraform.tfstate」は、Terraform が最後に apply した状態が記録されています。Terraform のコードと実際にプロビジョニングされたリソースの状態をマッピングできます。

※出力の一部をマスクしています。

```
$ cat terraform.tfstate
```

```
{
  "version": 4,
  "terraform_version": "1.5.7",
  "serial": 6,
  "lineage": "477718e5-e06e-7336-e338-b7a009a635dd",
```

```

"outputs": {},
"resources": [
  {
    "mode": "managed",
    "type": "oci_objectstorage_bucket",
    "name": "objectstorage_bucket",
    "provider": "provider[\"registry.terraform.io/oracle/oci\"]",
    "instances": [
      {
        "schema_version": 0,
        "attributes": {
          "access_type": "NoPublicAccess",
          "approximate_count": "0",
          "approximate_size": "0",
          "auto_tiering": "Disabled",
          "bucket_id": "ocid1.bucket.oc1.xx-xxxxx-xx.",
          "compartment_id": "ocid1.compartment.oc1..",
          "created_by": "ocid1.user.oc1..",
          "defined_tags": {},
          "etag": "5c39bd10-f705-48f6-a9ca-00ed480ac661",
          "freeform_tags": {},
          "id": "n/xxxxx/b/terraform",
          "is_read_only": false,
          "kms_key_id": null,
          "metadata": {},
          "name": "terraform",
          "namespace": "xxxxx",
          "object_events_enabled": false,
          "object_lifecycle_policy_etag": null,
          "replication_enabled": false,
          "retention_rules": [],
          "storage_tier": "Standard",
          "time_created": "2025-01-08 10:21:50.068 +0000 UTC",
          "timeouts": null,
          "versioning": "Disabled"
        },
        "sensitive_attributes": [],
        "private": "eyJlMmJmYjczMC1lY2FhLTExZTYtOGY4OC0zNDM2M2JjN2M0YzAiOnsiY3JlYXRlIjoxMjAwMDAwMDAwMDAwLCJkZWxldGUiojEyMDAwMDAwMDAwMDAsInVwZGF0ZSI6MTIwMDAwMDAwMDAwMDAwMH19"
      }
    ]
  }
]

```

```

    ]
  }
],
"check_results": null
}

```

ここで、再度適用を試みると、リソースが既に存在しているため、新たに作成されないことを確認できます。

※ `-auto-approve` オプションは、`apply` 時に `yes` の入力を省きます。

```
$ terraform apply -auto-approve
```

```
oci_objectstorage_bucket.objectstorage_bucket: Refreshing state... [id=n/orasejapan/b/terraform]
```

```
No changes. Your infrastructure matches the configuration.
```

```
Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.
```

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

プロビジョニングしたリソースを破棄します。

※出力の一部をマスクしています。

```
$ terraform destroy
```

```
oci_objectstorage_bucket.objectstorage_bucket: Refreshing state... [id=n/orasejapan/b/terraform]
```

```
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
```

```
- destroy
```

```
Terraform will perform the following actions:
```

```
# oci_objectstorage_bucket.objectstorage_bucket will be destroyed
- resource "oci_objectstorage_bucket" "objectstorage_bucket" {
  - access_type           = "NoPublicAccess" -> null
  - approximate_count     = "0" -> null
  - approximate_size      = "0" -> null
  - auto_tiering           = "Disabled" -> null
  - bucket_id             = "ocid1.bucket.oc1.xx-xxxxx-xx." -> null
  - compartment_id        = "ocid1.compartment.oc1.." -> null
  - created_by            = "ocid1.user.oc1.." -> null
  - defined_tags           = {} -> null
  - etag                  = "5c39bd10-f705-48f6-a9ca-00ed480ac661" -> null
  - freeform_tags         = {} -> null
  - id                    = "n/xxxxx/b/terraform" -> null
  - is_read_only          = false -> null
  - metadata              = {} -> null
  - name                  = "terraform" -> null
  - namespace             = "xxxxx" -> null
  - object_events_enabled = false -> null
  - replication_enabled   = false -> null
  - storage_tier           = "Standard" -> null
  - time_created          = "2025-01-08 10:21:50.068 +0000 UTC" -> null
  - versioning            = "Disabled" -> null
}
```

Plan: 0 to add, 0 to change, 1 to destroy.

Do you really want to destroy all resources?

Terraform will destroy all your managed infrastructure, as shown above.

There is no undo. Only 'yes' will be accepted to confirm.

「yes」と入力して実行します。

Enter a value: yes

```
oci_objectstorage_bucket.objectstorage_bucket: Destroying... [id=n/orasejapan/b/
terraform]
```

```
oci_objectstorage_bucket.objectstorage_bucket: Destruction complete after 2s
```

破棄されたかを確認します。

```
$ terraform show
```

```
The state file is empty. No resources are represented.
```

このような流れで、クラウドリソースのプロビジョニングを実行できます。

第3章のOCI Resource Managerを利用すれば、これらのTerraform操作をOCIのマネージドサービスとして実行できます。OCI Resource ManagerはTerraformの計画や適用をクラウド上で管理し、リソースのプロビジョニングを簡略化します。例えば、ローカルでTerraformをインストールする必要がなく、OCI Console Dashboardからスタック（Terraform構成）をアップロードして直接適用できます。また、OCI固有の機能（認証設定の簡略化やログの統合管理など）が組み込まれているため、OCI環境でのTerraform運用をさらに効率的に行えます。

OCI Kubernetes Engine 運用基礎知識

第3部のハンズオンでは、OCI Kubernetes Engine (OKE) をコンテナアプリケーションの基盤として利用しました。Kubernetesは、定期的にマイナーバージョンやメジャーバージョンのアップデートが行われます。運用においては、これらのアップデートへの対応が不可欠です。

オープンソースソフトウェア (OSS : Open Source Software) としてのKubernetes自体の継続的なアップデートに加え、それに追従する形でクラウドサービスプロバイダが提供するマネージドサービスでも、各プロバイダ独自の方法でクラスタアップデートが行われます。

ここでは、OSSにおけるKubernetesのアップデートの考え方と、OKEにおける具体的なクラスタアップデートの方法について解説します。

オープンソースソフトウェア

(OSS : Open Source Software) のKubernetes アップデート

KubernetesはOSSとして、以下のポリシーに基づいてアップデートがリリースされます。

≫ 1. リリースサイクル

Kubernetesは約4か月ごとに新しいマイナーバージョンがリリースされます（年3回）。メジャーバージョン（例：v1.xからv2.x）は現在のところリリースされていませんが、将来的に変更される可能性があります。

≫ 2. サポート期間

Kubernetesは3つのマイナーリリースバージョンをサポートします。例えば、現行の最新バージョンが1.32の場合は、1.32、1.31、1.30がサポートされます。最新のバージョンを利用することで、安定性やセキュリティを確保できます。

≫ 3. クラスタアップデートの計画と実施

OSSのKubernetesを直接運用する場合、クラスタ管理者が行う作業を表-17に整理します。

作業	概要
事前準備	アップデート前に互換性やリリースノートを確認し、既存の構成やアプリケーションへの影響を評価する。
Control Planeのアップデート	Control Planeのコンポーネントを新しいバージョンにアップデート。
Worker Nodeのアップデート	Nodeのコンポーネントを新しいバージョンにアップデート。

表-17 クラスタ管理者が行う作業

Worker Nodeのアップデートでは、クラスタの安定性を保つために、Kubernetesに実装されている仕組みを活用します。

まず、Cordon（コマンド：kubectrl cordon node）を実行して対象ノードへの新規Podのスケジューリングを停止し、既存のPodのみが動作する状態にします。

次に、Drain（コマンド：kubectrl drain node）により、そのノード上のPodを他のノードへ退避させます。この再スケジューリングはKubernetesのスケジューラーによって自動的に行われ、ノードを安全にアップデートできる状態にします。

最後に、ローリングアップデートを用いてノードを順番にアップデートすることで、クラスタ全体の可用性を維持しながらアップデートを進めます。

これらの作業は、OSSのKubernetesではkubectrlコマンドを用いて手動で実施する必要がありますが、OKEではこの一連の処理が自動化されており、ユーザーがkubectrlコマンドを実行する必要はありません（後述する、既存のノード・プールを新しいノード・プールに置き換える場合は除く）。

OSSにおけるKubernetesのクラスタアップデートの概要を理解したところで、次にOKEにおけるクラスタアップデートの方法について説明します。

OKEのクラスタアップデートについて

OKEは、Kubernetesのマネージドサービスとして、クラスタのアップデートを簡素化し、運用負荷を軽減する機能を提供します。OKEでは、基本クラスタ (Basic Cluster) と拡張クラスタ (Enhanced Cluster) という2種類のクラスタタイプ (表-02) と管理対象ノード (Managed Nodes) と仮想ノード (Virtual Nodes) という2種類のノードタイプ (表-03) があります。

特徴	基本クラスタ (Basic Cluster)	拡張クラスタ (Enhanced Cluster)
概要	基本クラスタは、KubernetesおよびKubernetes Engineによって提供される全てのコア機能を提供。	拡張クラスタは、基本クラスタで提供されていない以下の拡張機能をサポート。主な機能のみ。(執筆時点) ● 仮想ノード (Virtual Nodes) ● クラスタ・アドオン機能 ● Workload Identity ● オンデマンドノードサイクル
サポートするWorker Node数	1000 ノードまで	5000 ノードまで (flannel CNI プラグイン 商用レム) 2000 ノード (OCI VCN-native Pod Networking CNI プラグイン)
SLA (Service Level Agreement)	なし	あり (単一 Availability Domain : 99.9% / 複数 Availability Domain : 99.95%)

※ [3-2 OCI Kubernetes Engine] 表3-3の再掲

表-18 基本クラスタ (Basic Cluster) と 拡張クラスタ (Enhanced Cluster) の特徴

特徴	管理対象ノード (Managed Nodes)	仮想ノード (Virtual Nodes)
概要	コンピュート・インスタンス (ベア・メタルまたは仮想マシン) で実行され、ユーザーが一部管理する必要があるノード。	Oracleが完全に管理するサーバーレスで、インフラを意識せず利用できるノード。
アップデート作業	後述する3つの方法でアップデート。一部自動および完全手動の方法がある。	ユーザーによる作業は、管理コンソールからアップデートの実行をボタンをクリックするだけで、Control PlaneおよびWorker Nodeを自動アップデート。(ノードの追加削除、CordonやDrain作業も含む)
スケーリング	手動または自動ノードスケーリングが可能。	手動ノードスケーリング。自動ノードスケーリングは未対応。(執筆時点)
ユースケース	ノードにSSH接続してカスタマイズするなど自由度を高めてノードを管理する場合に適する。	ノードの管理が不要で、運用負荷を削減する場合に適する。

表-19 管理対象ノードと仮想ノードの特徴

OKEでのクラスタアップデートは以下の特徴を持っています。

≫ 1. マネージドなアップグレード機能

- OKEではControl PlaneとWorker Nodeのアップグレードを個別に管理。
- Oracleが新しいKubernetesバージョンをサポートすると、OCI Console DashboardのOKE管理画面やCLIを通じてバージョンアップのオプションが利用可能になる。

≫ 2. サポートされるバージョンとスケジュール

OKEでは、3つのマイナーリリースバージョンをサポートします。例えば、現行の最新バージョンが1.32の場合は、1.32、1.31、1.30がサポートされます。最新のバージョンを利用することで、安定性やセキュリティを確保できます。詳細は、公式ドキュメント^{※15}を参照してください。

≫ 3. OKEのクラスタアップデート方法

OKEでは、Control PlaneとWorker Nodeのアップデートを個別に実施します。Control Planeは、新規バージョンがリリースされると管理コンソールから手順に従ってアップデートを実行できます。Worker Nodeアップデートは、以下の3つの方法が提供されています。

ただし、Worker Nodeについては、基本クラスタ (Basic Cluster)、拡張クラスタ (Enhanced Cluster)、管理対象ノード (Managed Nodes)、仮想ノード (Virtual Nodes) で利用できる方法とできない方法があります。まずは、3つのアップデート方法の特徴を整理します。

1. オンデマンドノードサイクル

- ノード・プール内の既存のノードを順次新しいバージョンに自動で置き換える。
- ユーザーの開始をトリガーにOKEが自動的にインプレース・アップグレード。CordonおよびDrain作業も自動実行。

2. 既存のノード・プールのノード手動置換

- 管理者が既存のノードを手動で削除。
- 新しいバージョンのノードが自動追加。CordonおよびDrain作業も自動実行。

※15 公式ドキュメント:「サポートされているKubernetesのバージョン」https://docs.oracle.com/ja-jp/iaas/Content/ContEng/Concepts/contengaboutk8sversions.htm#contengaboutk8sversions_topic-Planned_Kubernetes_Versions_Support

- 既存ノードの削除は、ノードの数だけ手動で実施するため、数が多い場合はCLIでスクリプト化することを推奨。

3. 既存のノード・プールを新しいノード・プールに置換

- 現在のノード・プールを新しいバージョンで構成された別のノード・プールに手動で置き換える。
- 新規ノード・プールを手動で作成。CordonおよびDrain作業は自動実行。
- 手動作業の範囲が広いと、1または2のアップデート方法よりも運用負荷がかかる。

クラスタタイプとノードタイプ別に利用できるOKEのクラスタアップデート方法を表-20に整理します。

クラスタアップデート方法	基本クラスタ (Basic Cluster)	拡張クラスタ (Enhanced Cluster)	
	管理対象ノード (Managed Nodes)	管理対象ノード (Managed Nodes)	仮想ノード (Virtual Nodes)
オンデマンドノードサイクル	×	○	—
既存のノード・プールのノード手動置換	○	○	—
既存のノード・プールを新しいノード・プールに置換	○	○	—

※仮想ノード (Virtual Nodes) は、完全マネージドのため、クラスタアップデート作業が不要。(OCI Console Dashboardからアップデートをトリガーするのはユーザー)

表-20 タイプ別OKEクラスタアップデート対応表

表-20の通り、OKEのクラスタアップデートでは、拡張クラスタ (Enhanced Cluster) の管理対象ノード (Managed Nodes) におけるオンデマンドノードサイクルと仮想ノード (Virtual Nodes) が運用負荷を抑えて実施できる方法です。

>> 4. OKEのクラスタアップデート実践

OKEのクラスタアップデートの実践として、拡張クラスタ (Enhanced Cluster) の管理対象ノード (Managed Nodes) におけるオンデマンドノードサイクルを説明します。

オンデマンドノードサイクルは、ユーザーが指定したパラメータに応じてPodのスケジューリング制御やPodの待避を自動的に行います。また、ユーザーのアップグレード戦略に応じて、新バージョンノードのプロビジョニングや旧バージョンノードの削除も実施します。なお、オンデマンドノードサイクルでは、Worker Nodeのアップグレードを既存のWorker

Nodeをアップグレードするのではなく、新しいバージョンのWorker Nodeをプロビジョニングすることで実現します。

オンデマンドノードサイクルでユーザーが指定できるパラメータは、表-21の通りです。

パラメータ	概要
最大サージ	バージョンアップ時に一度に追加するノード数の上限。
最大使用不可	バージョンアップ時に使用できないようにするノード数の上限。

表-21 オンデマンドノードサイクルのパラメータ概要

バージョンアップ時にサービスダウンを限りなくゼロにした場合は、最大サージ数を 1 以上にして、最大使用不可を 0 にします。この場合は一時的にノード数が増えますので、その分の一時コストがかかることになります。

逆に、多少のサービスダウンを許容しても、バージョンアップ時にコストを増やしたくないのであれば、最大サージ数を0にして、最大使用不可を1以上にします。この場合は一時的にノード数が減りますのでスケジューリングできないPod（コンテナ）が発生し、サービスダウンが発生する可能性があります（図-32）。

後半で、最大サージを 1、最大使用不可を 0 に設定したオンデマンドノードサイクルを実践例としてスクリーンショットを交えて説明します。

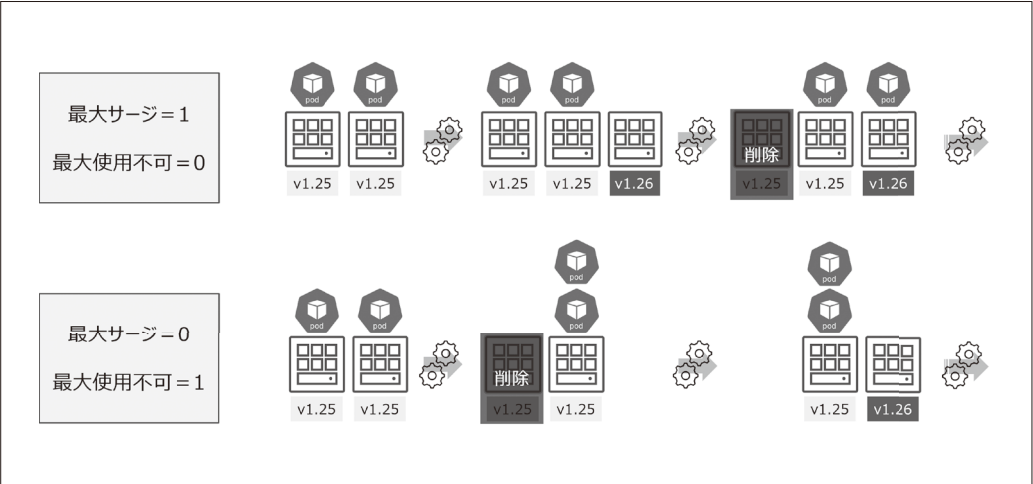


図-32 オンデマンドノードサイクルの動き

オンデマンドノードサイクルを実践します。使用する環境は表-22の通りです。

クラスタタイプ	ノードタイプ	ノード数	クラスタバージョン
拡張クラスタ (Enhanced Cluster)	管理対象ノード (Managed Nodes)	2	v1.30.1 ⇒ v1.31.1

表-22 OKEクラスタアップデート使用環境

クラスタアップデートの手順は、以下の通りです。

1. Control Planeのアップデート
2. ノード・プールのアップデート
3. オンデマンドノードサイクルの設定（最大サージと最大使用不可）
4. オンデマンドノードサイクルの実行

>> 1. Control Planeのアップデート

OCI Console Dashboardから「Kubernetesの新バージョンを使用できます」をクリックします（図-33）。



図-33 Control Planeのアップデート1

「Kubernetesバージョン」で「v1.31.1」を選択して、「アップグレード」ボタンをクリックします（図-34）。

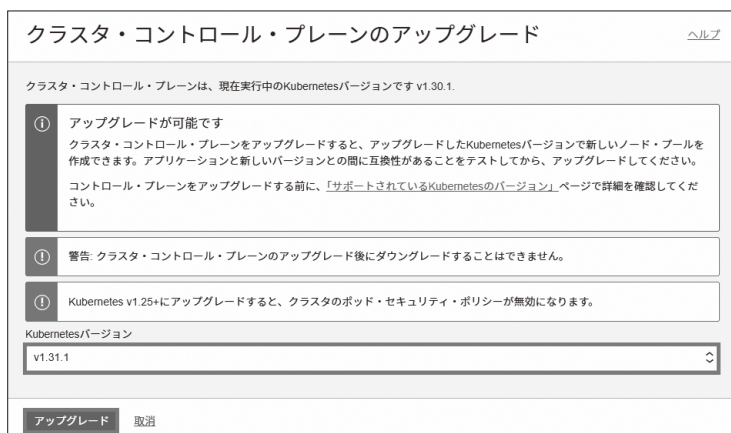


図-34 Control Planeのアップデート2

「更新中」の画面になります(図-35)。



図-35 Control Planeのアップデート3

しばらくすると、「更新中」から「アクティブ」に変わり、Kubernetesのバージョンが v1.30.1 から v1.31.1 に変わります(図-36)。



図-36 Control Planeのアップデート4

これでControl Planeのアップデート作業は完了です。この時点では、まだノード・プールとWorker Nodeはアップデートされていません。

>> 2. ノード・プールのアップデート

OCI Console Dashboardで「クラスタ詳細」から「ノード・プール」に移り、対象のノード・プール名、ここでは「pool1」をクリックします (図-37)。

コンパートメント内のノード・プール					
ノード・プールの一覧					
ノード・プール	ノード・タイプ	ノード・プールのステータス	Kubernetesバージョン	イメージ名	合計ワーカー・ノード
pool1	managed	● アクティブ	▲ v1.30.1 ①	Oracle-Linux-8-90-2024-09-30-0-CKE-1.30.1-747	2

図-37 ノード・プールのアップデート1

「編集」ボタンをクリック後、「バージョン」で「v1.31.1 (現在のクラスタ・バージョン)」を選択して、「変更の保存」ボタンをクリックします (図-38)。

コンテナ・クラスタ

クラスタ詳細

ノード・プールの詳細

pool1

編集

サイクル・ノード

ノードのトラブルシューティング

ノード・プールの詳細

Kubernetesラベル

ノード・プールのステータス ● アクティブ

ノード・プールID: mw74upa 表示 コピー

ノード・タイプ: 管理対象

コンパートメント:

ネットワーク・セキュリティ・グループ: なし

OCPU: 1

メモリー量 (GB): 16

配置構成

AD 1: TQA.UK.LONDON-1-AD-1

ワーカー・ノード・サブネット: oke-nodesubnet-quick-cluster1-39c5d7a78-regional

管理タイプ: オンデマンド容量

ポッド通信

サブネット: oke-nodesubnet-quick-cluster1-39c5d7a78-regional

ネットワーク・セキュリティ・グループ: なし

1ノード当たりのポッド数: 31

リソース

ノード

ノード・プールの編集

ノード・プールの一定の特性を編集できます。

名前

pool1

ノード・プールの新しい名前

バージョン

v1.31.1 (現在のクラスタ・バージョン)

このノード・プールに適用されたノードで、アップグレード・バージョンが実行されます

ノード数 ①

2

ノード配置構成

可用性ドメイン ①

TQA.UK.LONDON-1-AD-1

のワーカー・ノード・サブネット ① (コンパートメントの変更)

oke-nodesubnet-quick-cluster1-39c5d7a78-regional (リジヨナル)

フォルト・ドメイン オプション

フォルト・ドメインの選択

拡張オプションの表示

可用性ドメイン ①

TQA.UK.LONDON-1-AD-2

のワーカー・ノード・サブネット ① (コンパートメントの変更)

oke-nodesubnet-quick-cluster1-39c5d7a78-regional (リジヨナル)

フォルト・ドメイン オプション

フォルト・ドメインの選択

拡張オプションの表示

変更の保存 取消

図-38 ノード・プールのアップデート2

ノード・プールのバージョンがKubernetes v1.31.1 に変わります (図-39)。



図 -39 ノード・プールのアップデート 3

これでノード・プールのアップデートは完了です。この時点では、まだWorker Nodeはアップデートされていません。

≫ 3. オンデマンドノードサイクルの設定（最大サージと最大使用不可）

ここまでの手順は、基本クラスタ（Basic Cluster）と管理対象ノード（Managed Nodes）でも同じです。

そしてここから実践するのが、拡張クラスタ（Enhanced Cluster）と管理対象ノード（Managed Nodes）で実行できるオンデマンドノードサイクルです。

OCI Console Dashboardから「サイクル・ノード」ボタンをクリックします（図-40）。



図 -40 オンデマンドノードサイクルの設定（最大サージと最大使用不可） 1

図-32と同じ設定として、最大サージを1、最大使用不可を0に設定します。そして、「サイクル・ノード」ボタンをクリックします（図-41）。

サイクル・ノード
ヘルプ

ノード・プールpool1で起動する新しいワーカー・ノードは、現在のノード・プール・プロパティで作成されます。ノード・プールの更新をご使用のノード・プールに適用しますか。これにより、既存のすべてのノードが、現在のノード・プール・プロパティを実行しているノードに置き換えられます。

最大サイズ(追加ノードの最大数またはパーセンテージ)
☒ 整数 ☐ パーセンテージ

最大使用不可(使用不可ノードの最大数またはパーセンテージ)
☒ 整数 ☐ パーセンテージ

サイクル・ノード 取消

図-41 オンデマンドノードサイクルの設定（最大サージと最大使用不可） 2

v1.30.1の2ノードに加えて、v1.31.1ノードが追加されます（図-42）。ここが、図-32で説明したノードが一時的に3ノードとなることで一時的なコストが発生する箇所になります。

ノード名	プライベートIP	Kubernetesのノード条件 ①	ノード状態 ①	Kubernetesバージョン
oke-ctfuv74n5q-n3nmwk74upa-stweaggozca-0	10.0.10.172	● 準備完了	● アクティブ	▲ v1.30.1 ① ①
oke-ctfuv74n5q-n3nmwk74upa-stweaggozca-1	10.0.10.118	● 準備完了	● アクティブ	▲ v1.30.1 ① ①
oke-ctfuv74n5q-n3nmwk74upa-stweaggozca-3	使用不可	● 使用不可	● 作成中	● v1.31.1

3アイテムを表示中 < 1/1 >

図-42 オンデマンドノードサイクルの設定（最大サージと最大使用不可） 3

その後、v1.31.1ノードがアクティブになると、v1.30.1の1ノードが削除されます。削除後、新たにv1.31.1ノードが追加されます（図-43）。

ノード名	プライベートIP	Kubernetesのノード条件 ①	ノード状態 ①	Kubernetesバージョン
oke-ctfuv74n5q-n3nmwk74upa-stweaggozca-1	10.0.10.118	● 準備完了	● アクティブ	▲ v1.30.1 ① ①
oke-ctfuv74n5q-n3nmwk74upa-stweaggozca-3	10.0.10.213	● 準備完了	● アクティブ	● v1.31.1
oke-ctfuv74n5q-n3nmwk74upa-stweaggozca-2	使用不可	● 使用不可	● 作成中	● v1.31.1

3アイテムを表示中 < 1/1 >

図-43 オンデマンドノードサイクルの設定（最大サージと最大使用不可） 4

v1.31.1ノードがアクティブになると 最後のv1.30.1ノードが削除されて、v1.31.1の2ノードになります（図-44）。

ノード				
ノード名	プライベートIP	Kubernetesのノード条件 ①	ノード状態 ①	Kubernetesバージョン
oke-ctrln74n5q-n3nnw674upa-slveagpozca-3	10.0.10.213	● 準備完了	● アクティブ	● v1.31.1
oke-ctrln74n5q-n3nnw674upa-slveagpozca-2	10.0.10.246	● 準備完了	● アクティブ	● v1.31.1
				2アイテムを表示中 < 1 / 1 >

図-44 オンデマンドノードサイクルの設定（最大サージと最大使用不可） 5

図-32で説明した通りの動作になることを確認できました。オンデマンドノードサイクル以外の方法では、ノード・プールのアップデート後に新バージョンのノード追加、CordonおよびDrain作業（オプションで自動実行可能）、旧バージョンのノード削除を手動で行う必要があります。オンデマンドノードサイクルを利用することで、この手動作業の負荷を低減できます。

最後に、拡張クラスタ（Enhanced Cluster）と仮想ノード（Virtual Nodes）のクラスタアップデートについて補足します。

オンデマンドノードサイクルの最初の手順「1. Control Planeのアップデート」で、「Kubernetesの新バージョンを使用できます」というボタンをクリックして、最新バージョンを選択後、「アップグレード」ボタンをクリックしました。オンデマンドノードサイクルでは、この作業でControl Planeだけがアップデートされます。しかし、拡張クラスタ（Enhanced Cluster）と仮想ノード（Virtual Nodes）のクラスタアップデートでは、この作業だけで、Control Plane、ノード・プール、ノードの全てを自動アップデートします。

ここでは、OKEクラスタアップデートの方法を整理しました。ノードレベルでの整理に留まっており、実際にアプリケーションが稼働する環境ではありません。実際にアプリケーションが稼働する環境でのクラスタアップデートは、アプリケーションへの影響を含めた検証が必須となりますので、実利用では十分な検証を実施した上での利用を推奨します。

OpenTelemetryとOCI APMによるトレースとログの相関

オブザーバビリティの基本となる「メトリクス」「ログ」「トレース」は、それぞれを個別に見るのではなく、相互に関連付けて確認することが重要です。第6章のハンズオンでは、OCI APMを使い、トレース情報からJMXメトリクスを確認することで、「メトリクスとトレースの相関」を体験しました。ここでは、OpenTelemetryとOCI APMを活用し、「トレースとロ

グの相関」を実践します。

前提知識

「トレースとログの相関」の実践に入る前に、前提知識を確認しておきましょう。

≫ OpenTelemetry とロギング

OpenTelemetry の公式ドキュメント「OpenTelemetry Logging」^{※16}では、次のように記載されています。

If the recorded logs contained trace context identifiers (such as trace and span ids or user-defined baggage) it would result in much richer correlation between logs and traces, as well as correlation between logs emitted by different components of a distributed system. This would make logs significantly more valuable in distributed systems.

つまり、ログメッセージにトレースコンテキスト識別子（例えば、トレース ID、スパン ID）が含まれていれば、トレースとログを関連付けて確認できるようになります。これにより、分散システム全体にわたるログ間の関連性も高まり、「トレースとログの相関」が実現されます。

≫ OpenTelemetry とロギングの言語サポート

OpenTelemetry では、主要なプログラミング言語に対応したライブラリが提供されており、ロギング機能についても各言語ごとのサポート状況が公式ドキュメント^{※17}で公開されています。執筆時点での対応状況は、表-23の通りです。

ステータスは、「Stable（安定版）」「Beta（ベータ版）」「Development（開発中 / アルファ版）」の3段階で示されています。

※16 OpenTelemetry 公式ドキュメント「OpenTelemetry Logging」<https://opentelemetry.io/docs/specs/otel/logs/>

※17 OpenTelemetry 公式ドキュメント「言語サポート」<https://opentelemetry.io/ja/docs/concepts/signals/logs/>

言語	ステータス
C++	Stable
C#/NET	Stable
Erlang/Elixir	Development
Go	Beta
Java	Stable
JavaScript	Development
PHP	Stable
Python	Development
Ruby	Development
Rust	Beta
Swift	Development

表-23 OpenTelemetry Loggingのサポート状況

>> Mushopでの実装例

ハンズオンで使用しているサンプルアプリケーションMuShopにおいて、OpenTelemetry Loggingの実装方法を見ていきましょう。今回は、Helidonを使用して構築されたcartsアプリケーションを例に、その実装内容を確認します。

JavaにおけるOpenTelemetry Loggingでは、一般的にLogbackというロギングライブラリがよく利用されており、本アプリケーションでもLogbackを採用しています。ただし、Helidonはデフォルトでjava.util.loggingを使用しているため、Logbackを利用するには切り替えのための追加実装が必要です。また、Logbackを導入するにあたり、ロギングのファサード^{※18}としてSLF4Jも併用します。

cartsアプリケーションでは、以下のようなロギング関連の依存関係をpom.xmlに定義しています。

```
<dependency>
  <groupId>io.opentelemetry.instrumentation</groupId>
  <artifactId>opentelemetry-logback-appender-1.0</artifactId>
  <version>2.14.0-alpha</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
```

※18 GoF (Gang of Four) によるデザインパターンの一つである「Facadeパターン」。複雑な実装の背後に統一されたインターフェースを提供することで、Javaにおけるロギングの実装を柔軟に切り替え可能。

```

    <artifactId>slf4j-jdk-platform-logging</artifactId>
    <version>2.0.7</version>
  </dependency>
  <dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jul-to-slf4j</artifactId>
  </dependency>

```

それぞれの役割は表-24の通りです。

ライブラリ	役割
opentelemetry-logback-appender-1.0	OpenTelemetry を利用した Logback のログ出力
slf4j-jdk-platform-logging	Java の標準 Logging API (System.Logger) と SLF4J を統合したロギングの実装
logback-classic	Logback 向けの SLF4J API の実装
jul-to-slf4j	java.util.logging と SLF4J の拡張実装

表-24 ロギングライブラリの役割

これらのライブラリを利用するために、「src/main/resources」配下に logback.xml を作成し、次のように定義します。

```

<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern><![CDATA[%date{HH:mm:ss.SSS} [%thread] %-5level %logger{15}#%line
%X{req.requestURI} traceId: %X{trace_id} spanId: %X{span_id} %msg\n]]></pattern>
    </encoder>
  </appender>

  <appender name="OTEL" class="io.opentelemetry.instrumentation.logback.v1_0.
OpenTelemetryAppender">
    <appender-ref ref="STDOUT" />
  </appender>

```

```

<root>
  <level value="INFO" />
  <appender-ref ref="STDOUT" />
</root>

</configuration>

```

上記の pattern 設定にあるように、ログメッセージに traceId や spanId を出力することで、実行時のトレース情報とログの対応関係を明示的に記録できます。

実際のログ出力例は以下の通りです。

```

2025-04-10T08:35:42.126660771+00:00 stdout F 08:35:42.126 [pool-3-thread-1] INFO m.
FulfillmentService#149 traceId: f700955b941efdc6672f66542f90e675 spanId:
5dd4792bad791127 Sending shipment update {"orderId":4053,"shipment":{"id":"a5585
5c6-1c90-47a8-9d4c-bfd60a1c1afa","name":"Shipped"}}

```

このようにログにトレースIDやスパンIDを含めることで、「トレースとログの相関」を実現できます。なお、これは一例であり、他にも様々な実装方法が存在します^{※19}。

「トレースとログの相関」の実践

このハンズオンを始める前に、第6章の手順を全て完了している必要があります。

また、「トレースとログの相関」では、MuShopに含まれるJavaアプリケーション(carts、orders、fulfillments)を対象に進めていきます。

≫ OCI Logging Analytics の有効化とログ・グループの作成

「トレースとログの相関」では、OCI Logging Analytics^{※20}というサービスを利用してログ分析を実施します。

最初に、OCI Logging Analyticsの有効化処理を行います。ナビゲーションメニューから

※19 「Appender Instrumentation for Logback version 1.0 and higher」<https://github.com/open-telemetry/opentelemetry-java-instrumentation/blob/main/instrumentation/logback/logback-appender-1.0/library/README.md>

※20 OCI Logging Analytics は、OCI Logging に収集されたログを統合・分析するサービス。
公式ウェブサイト：<https://www.oracle.com/jp/manageability/logging-analytics/>

「監視および管理」を選択して、「管理」をクリックします (図-45)。



図-45 OCI Logging Analyticsの有効化とログ・グループの作成 1

「ログ・アナリティクスの仕様の開始」ボタンをクリックします (図-46)。

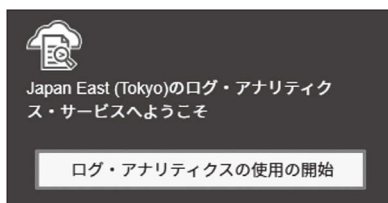


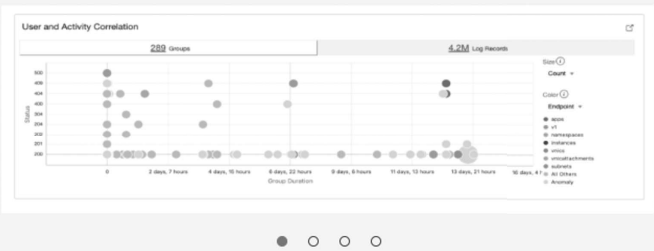
図-46 OCI Logging Analyticsの有効化とログ・グループの作成 2

ログ・アナリティクスの有効化処理が完了後、「次」ボタンをクリックします (図-47)。

ログ・アナリティクスの有効化

- ✓ ログ・アナリティクス・サービスは有効です。
- ✓ ポリシー"logging_analytics_automatic_service_policies"が作成されました。ポリシーには「アイデンティティ・サービス」からアクセスできます。
 - define tenancy sampledata as
 - endorse group Administrators to read loganalytics-features-family in tenancy sampledata
 - endorse group Administrators to read loganalytics-resources-family in tenancy sampledata
 - endorse group Administrators to read compartments in tenancy sampledata
 - allow service loganalytics to READ loganalytics-features-family in tenancy
 - allow service loganalytics to READ compartments in tenancy
- ✓ コンパートメント"..."にログ・グループ"Default"が作成されました。

次のステップ: データの追加



次

図-47 OCI Logging Analyticsの有効化とログ・グループの作成 3

「OCI 監査ログ収集の構成」で、「次」ボタンをクリックします (図-48)。



図-48 OCI Logging Analyticsの有効化とログ・グループの作成 4

さらに、「OCI監査ログ収集の構成」で、「閉じる」ボタンをクリックします (図-49)。

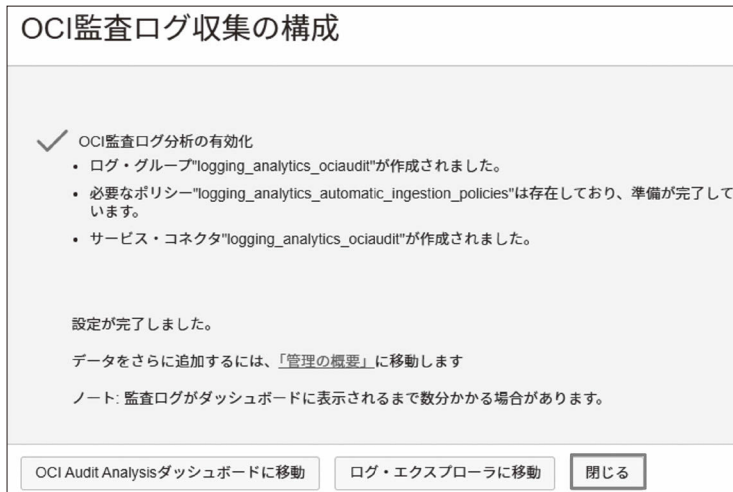


図-49 OCI Logging Analyticsの有効化とログ・グループの作成 5

画面が遷移して、「ログ・アナリティクス」のホームが表示されます (図-50)。

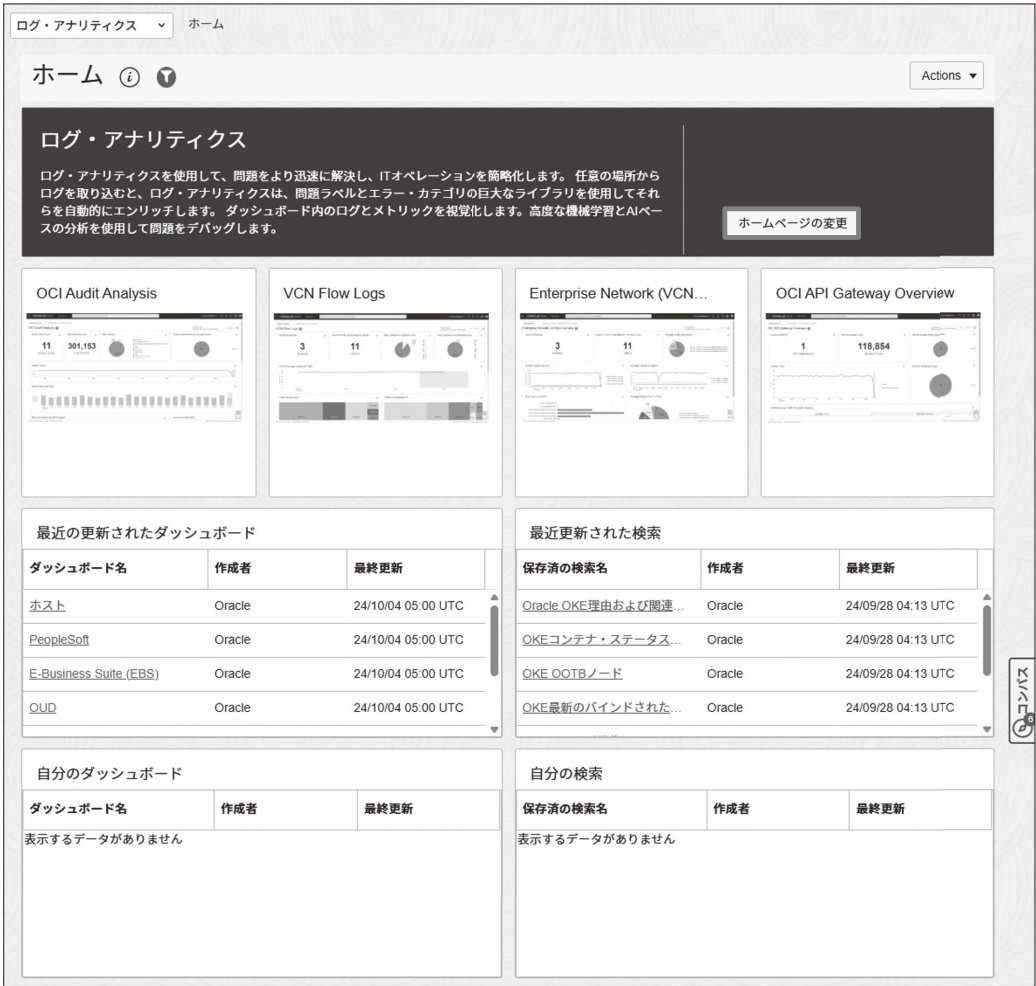


図 -50 OCI Logging Analyticsの有効化とログ・グループの作成 6

以上で、OCI Logging Analytics の有効化処理は完了です。

ここから、ログ・グループを作成します。画面左上にあるプルダウンメニューから「管理」を選択します (図-51)。

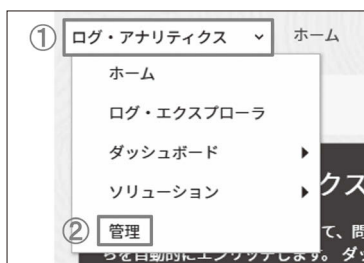


図-51 OCI Logging Analyticsの有効化とログ・グループの作成 7

「管理の概要」にある「ログ・グループ」の「ログ・グループの作成」テキストリンクをクリックします (図-52)。



図-52 OCI Logging Analyticsの有効化とログ・グループの作成 8

設定する内容は、表-25の通りです。設定後、「作成」ボタンをクリックします (図-53)。

設定項目	内容
コンパートメント	対象のコンパートメント
名前	mushop

表-25 「ログ・グループの作成」設定内容

ログ・グループの作成

コンパートメント 読取り専用

①

名前

mushop

説明 オプション

拡張オプションの表示

③

作成

取消

図-53 OCI Logging Analyticsの有効化とログ・グループの作成 9

作成後、ログ・グループの合計数が変わっています (図-54)。

管理の概要					
アラーム ① 0 合計	検出ルール 0 合計 ルールの作成	エンティティ 4 合計 エンティティの作成	ログ・グループ 3 合計 ログ・グループの作成	ソース 331 合計 ソースの作成	パーサー 278 合計 パーサーの作成
アップロード 0 合計 アップロード・ファイル	他の構成アイテム フィールド:1319 ラベル:100 保存済検索:91 ルックアップ:9 EMブリッジ:0	アクション 構成コンテンツのインポート 取得の設定	データの追加		

図-54 OCI Logging Analyticsの有効化とログ・グループの作成 10

以上で、ログ・グループの作成は完了です。

≫ Connector Hub を利用した OCI Logging と OCI Logging Analytics の連携

OCI Logging と OCI Logging Analytics を連携させるために、Connector Hub の設定を行います。ナビゲーションメニューから「監視および管理」をクリックし、「コネクタ」をクリックします (図-55)。

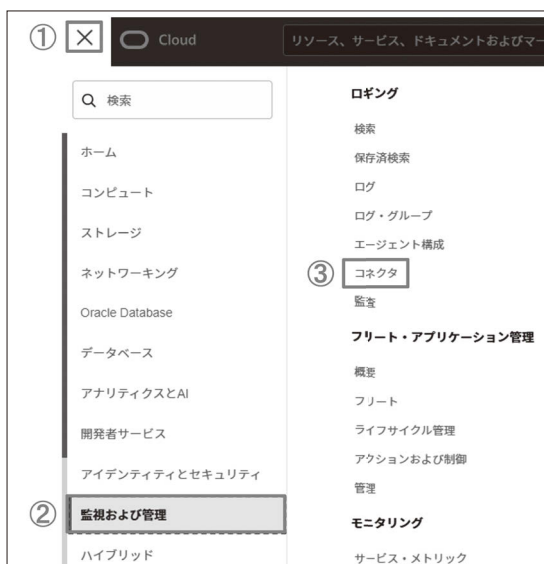


図-55 Connector Hubを利用したOCI LoggingとOCI Logging Analyticsの連携 1

フリートライアル において、デフォルトで作成できるコネクタ数は、2個までです。そのため、「logging_analytics_ociaudit」コネクタを削除します。メニューアイコンをクリックして、「削除」を選択します (図-56)。

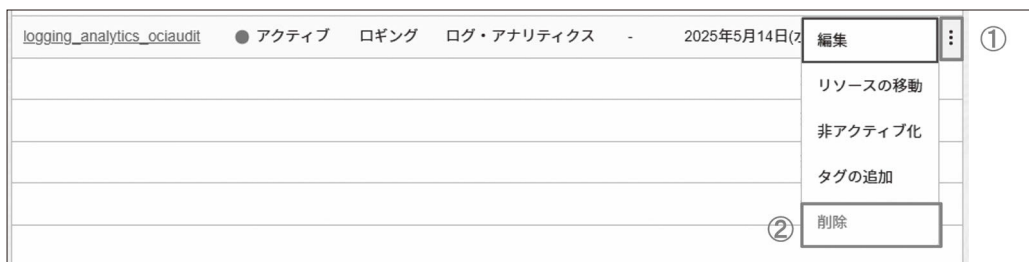


図-56 Connector Hubを利用したOCI LoggingとOCI Logging Analyticsの連携 2

「コネクタの作成」ボタンをクリックします (図-57)。



図-57 Connector Hubを利用したOCI LoggingとOCI Logging Analyticsの連携 3

設定する内容は、表-26の通りです。途中の茶色枠にある「このコネクタによるコンパートメント<対象のコンパートメント名>のモニタリングへの書込みを許可するデフォルト・ポリシーを作成します。」にある「作成」ボタンをクリックします (図-58)。

設定項目	内容
コネクタ名	mushop-appendix
リソース・コンパートメント※1	対象のコンパートメント
ソース	ロギング
ターゲット	ログ・アナリティクス
コンパートメント名※2	対象のコンパートメント
ログ・グループ (ソースの構成)	mushop
ログ	mushop
コンパートメント※3	対象のコンパートメント
ログ・グループ (ターゲットの構成)	mushop

※1：リソース・コンパートメントは、デフォルト値としてルートコンパートメントが指定されています。
※2：コンパートメント名は、デフォルト値としてルートコンパートメントが指定されています。
※3：コンパートメントは、デフォルト値としてルートコンパートメントが指定されています。

表-26 Connector Hubの設定内容

コネクタの作成

コネクタを作成して、ソース・サービスからターゲット・サービスへのデータ・フローを定義します。詳細は、コネクタ・ハブの概要を参照してください。

① **コネクタ名**
 mushop-appendix

② **説明**
 説明を入力

リソース・コンパートメント

③ **ソース**
 ログイン

④ **ターゲット**
 ログ・アナリティクス

コネクタの構成 ①

ソースの構成

⑤ **コンパートメント名**
 mushop

⑥ **ログ・グループ**
 mushop

⑦ **ログ**
 mushop

⑧ **ログ・フィルタ・タスク**
 (オプション)
 プロパティ 演算子 値
 ここで プロパティの選択 演算子の選択 値の選択

⑨ **ターゲットの構成**

⑩ **作成**

図-58 Connector Hubを利用したOCI LoggingとOCI Logging Analyticsの連携 4

「作成」ボタンをクリックします (図-59)。

作成 **取消**

図-59 Connector Hubを利用したOCI LoggingとOCI Logging Analyticsの連携 5

画面が遷移して、一覧が表示されます (図-60)。



図 -60 Connector Hubを利用したOCI LoggingとOCI Logging Analyticsの連携 6

これで、OCI LoggingからOCI Logging Analyticsにログが連携されます^{※21}。

以上で、Connector Hubを利用したOCI LoggingとOCI Logging Analyticsの連携設定は完了です。

※21 OCI LoggingやConector Hubを介さずにOKEからOCI Logging Analyticsに直接ログを転送可能。公式GitHub：
<https://github.com/oracle-quickstart/oci-kubernetes-monitoring/>

>> OCI Logging Analytics の設定

OCI Logging Analytics でトレースIDをもとに検索を行えるように設定します。OCI Logging Analytics のログ・グループを作成します。

ナビゲーションメニューから「監視および管理」をクリックし、「管理」をクリックします (図-61)。



図-61 OCI Logging Analytics の設定 1

画面左のリソースメニューから「ソース」のテキストリンクをクリックします (図-62)。

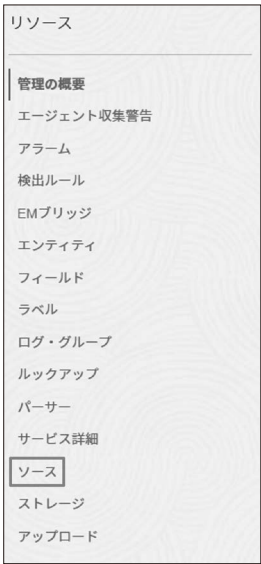


図 -62 OCI Logging Analyticsの設定 2

検索欄に「OCI Unified」と入力し、「OCI Unified Schema Logs」※22のテキストリンクをクリックします (図-63)。



図 -63 OCI Logging Analyticsの設定 3

画面遷移後、「編集」をクリックします (図-64)。



図 -64 OCI Logging Analyticsの設定 4

※22 OCI Unified Schema Logs は、OCI Logging Analytics でデフォルトで用意されている Linux ホスト向けの Oracle 定義のソース。

「拡張フィールド」タブを選択して、「追加」ボタンをクリックします (図-65)。

ソースの編集

名前 読取り専用

OCI Unified Schema Logs

説明 オプション 読取り専用

Logs coming in OCI Unified Schema format

ソース・タイプ 読取り専用

File

エンティティ・タイプ 読取り専用

Host (Linux)

パーサー 読取り専用

☐ 時間のみの自動解析 ⓘ
 ☒ 特定のパーサー ⓘ

デフォルト

カスタム

OCI Unified Schema Log Format ⓘ

含まれているパターン

除外されたパターン

データ・フィルタ

拡張フィールド

フィールド・エンリッチメント

ラベル

追加 ⓘ

サンプル・コンテンツで検索

ロック済	テストのステータス	ベース・フィールド	サンプル・コンテンツ	条件	拡張フィールド抽出式	有効
表示するデータがありません。						

ページ 1 (0/0アイテム) K < 1 > X

エージェント収集プロパティの表示

図-65 OCI Logging Analytics の設定 5

設定する内容は、表-27 の通りです。設定後、「テスト定義」ボタンをクリックして、「成功」と表示されていれば問題ありません。矢印のプルダウンを展開すると詳細を確認できます。「成功」を確認した後に、「追加」ボタンをクリックします (図-66)。

設定項目	内容
ベース・フィールド	Message
サンプルのベース・フィールド・コンテンツ	2025-04-10T08:35:42.126660771+00:00 stdout F 08:35:42.126 [pool-3-thread-1] INFO m.FulfillmentService#149 traceId: f700955b941efdc6672f66542f90e675 spanId: 5dd4792bad791127 Sending shipment update {"orderId":"4053","shipment":{ "id":"a55855c6-1c90-47a8-9d4c-bfd60a1c1afa","name":"Shipped"}}
式の抽出	traceId:\s+{Trace ID:\S+}\s+spanId:

表-27 「拡張フィールド定義の追加」の設定内容

拡張フィールド定義の追加

▶ 条件(オプション)

拡張フィールド定義

①

ベース・フィールド

Message

②

サンプルのベース・フィールド・コンテンツ

2025-04-10T08:35:42.126660771+00:00 stdout F 08:35:42.126 [pool-3-thread-1] INFO m.FulfillmentService#149 traceId: f700955b941efdc6672f66542f90e675 spanId: 5dd4792bad791127 Sending shipment update [{"orderId":4053,"shipment":{"id":"a55855c6-1c90-47a8-9d4c-bfd60a1c1afa"},"name":"Shipped"}]

③

式の抽出

traceId:'s'+(Trace ID:'s+')s+spanId:

④

テスト定義

テスト結果

一致ステータス

Trace ID

● 成功

f700955b941efdc6672f66542f90e675

詳細

Match Succeeded

ステップ数 68

サンプルのベース・フィールド・コンテンツ 2025-04-10T08:35:42.126660771+00:00 std f700955b941efdc6672f66542f90e675 spanId: 5dd4792bad791127 Sending shipment update [{"orderId":4053,"shipment":{"id":"a55855c6-1c90-47a8-9d4c-bfd60a1c1afa"},"name":"Shipped"}]

式の抽出 traceId:'s'+(Trace ID:'s+')s+spanId:

⑤

追加

取消

図-66 OCI Logging Analyticsの設定 6

「変更の保存」ボタンをクリックします (図-67)。

名前 読取り専用

OCI Unified Schema Logs

説明 オプション 読取り専用

Logs coming in OCI Unified Schema format

ソース・タイプ 読取り専用

File

エンティティ・タイプ 読取り専用

Host (Linux)

パーサー 読取り専用

☐ 時間のみ自動解析
 ☒ 特定のパーサー

デフォルト

カスタム

OCI Unified Schema Log Format

含まれているパターン

除外されたパターン

データ・フィルタ

拡張フィールド

フィールド・エンリッチメント

ラベル

追加

サンプル・コンテンツで検索

ロック済	テストのステータス	ベース・フィールド	サンプル・コンテンツ	条件	拡張フィールド抽出式	有効
	● 成功	Message	2025-04-10T08:35:42.126660771... stdout F 08:35:42.126 [pool-3-thread-1] INFO m.FulfillmentService#149 traceId: f700955b941efdc6672f6... spanId: 5dd4792bad791127 Sending shipment update {"orderId":"4053","shipm... {"id":"a55955c6-1c90-47a8-9d4c-bfd60a1c1afa"},"name"...	traceId:'s*(Trace ID:'s+)'s+spanId:	☒	⋮

ページ 1 / 1 (1/1アイテム)

エージェント収集プロパティの表示

変更の保存

取消

図-67 OCI Logging Analytics の設定 7

この設定により、ログ内のトレースIDをフィールドとして認識させることができます。
 以上で、OCI Logging Analytics の設定は完了です。

>> OCI APM から OCI Logging Analytics へのドリルダウン設定

OCI Logging Analytics でトレースIDをもとに検索を行えるように設定します。

問い合わせ URL の取得

OCI Logging Analytics のログ・グループを作成します。ナビゲーションメニューから「監視および管理」をクリックし、「ログ・エクスプローラー」をクリックします (図-68)。

75



図-68 問い合わせURLの取得 1

「OCI Unified Schema Logs」のテキストリンクをクリックするとメニューが表示されますので、「検索に追加」を選択します (図-69)。

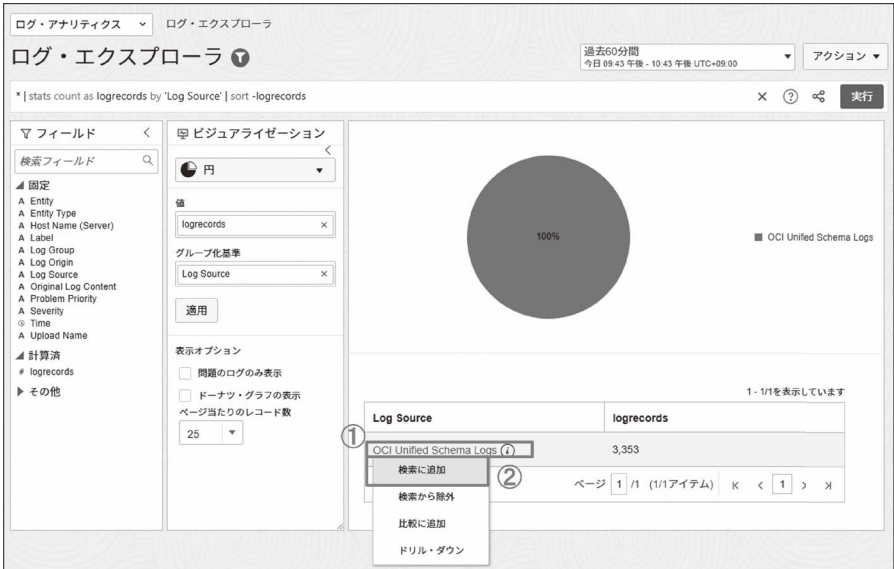


図-69 問い合わせURLの取得 2

「ビジュアライゼーション」のプルダウンをクリックし、「レコード」を選択します (図-70)。

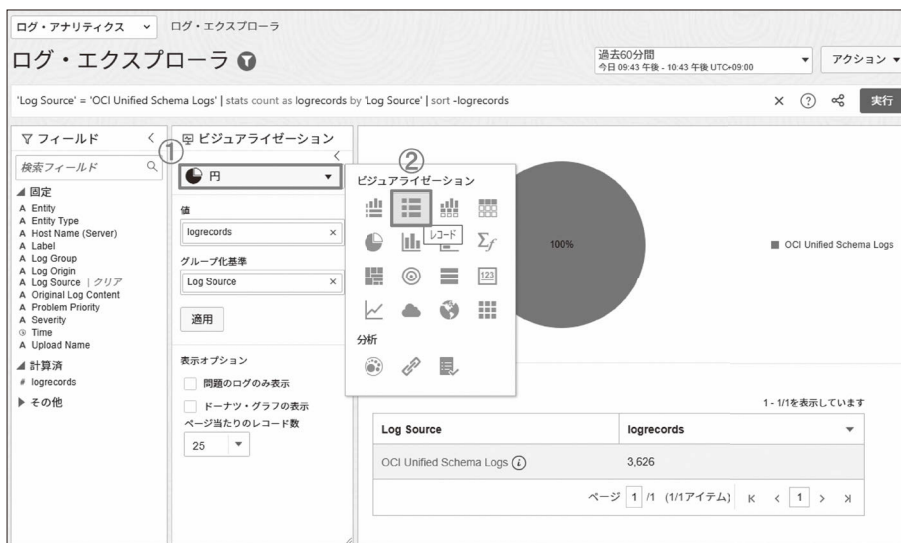


図-70 問い合わせURLの取得 3

「その他」から「Trace ID」をクリックします (図-71)。

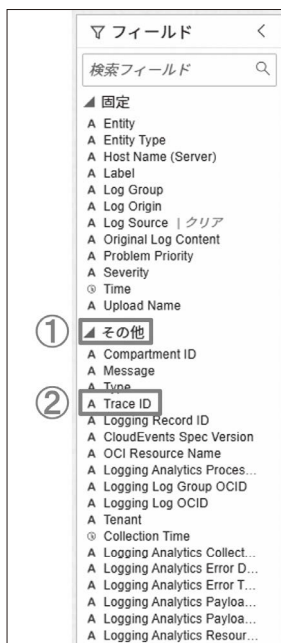


図-71 問い合わせURLの取得 4

表示されたトレースIDのいずれかにチェックを入れて、「適用」ボタンをクリックします (図-72)。



図-72 問い合わせURLの取得 5

右上にある赤枠部分をクリックし、問い合わせURLをクリップボードにコピーします (図-73)。



図-73 問い合わせURLの取得 6

以下のようなダイアログが表示されれば問題ありません (図-74)。

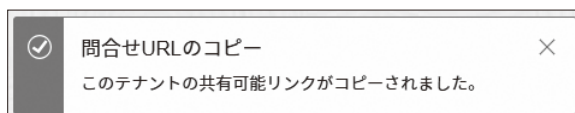


図-74 問い合わせURLの取得 7

コピーした文字列をメモ帳またはテキストエディタにペーストして保存しておいてください。

以下に例を示します。

<https://cloud.oracle.com/loganalytics/explorer?viz=records&encodedQuery=J0xvYzBTb3VyY2UwID0gJ09SSSBvbmlmaWVkaGFvYSBmb2dzJyBhbmgJ1RjYWwNIEIeJyA9IGQwMDA0GViYzc4NWl1ZGJ0DEZOTZlYTQ3NTAxNDYy&vizOptions=%7B%22customVizOpt%22%3A%7B%22primaryFieldName%22%3A%22body%22%2C%22primaryFieldDName%22%3A%22Original%20Log%20Content%22%7D%7D&scopeFilters=lg%3Aroot%2Ctrued%3Brse%3Aroot%2Ctrued%3Brg%3Aap-tokyo-1&timeNum=15&timeUnit=MINUTES®ion=ap-tokyo-1&tenant=xxxxxxxxxx>

この URL 内の Base64 encode 済みの「encodeQuery」から「&vizOptions」までの「encodedQuery=J0xvZyBTb3VyY2UnID0gJ09DSSBVbmlmaWVkaIFNjaGVtYSBMb2dzJyBhbmQgJ1RyYWNIIEIEJyA9IGQwMDA4OGViYzc4NWII1ZGJjODE2OTZlYTQ3NTAxN2Ey」部分を以下のように置き換えます。

```
query=%27Log+Source%27+%3D+%27OCI+Unified+Schema+Logs%27+and+%27Trace+ID%27+%3D+<Trace
Id>
```

これはBase64 decodeを行い、トレースIDをOCI APMが動的にバインドできるように「<TraceId>」という文字列を設定しています。ちなみに、このクエリをURLデコードすると以下ようになります。

```
query='Log Source' = 'OCI Unified Schema Logs' and 'Trace ID' = <TraceId>
```

これはLogging Analyticsのログ・エクスプローラーでの検索クエリになります。

URL全体のイメージは以下です。

<https://cloud.oracle.com/loganalytics/explorer?viz=records&query=%27Log+Source%27+%3D%270CI+Unified+Schema+Logs%27+and+%27Trace+ID%27+%3D<TraceId>&vizOptions=%7B%22customVizOpt%22%3A%7B%22primaryFieldIname%22%3A%22body%22%2C%22primaryFieldName%22%3A%22Original%20Log%20Content%22%27D%27D&scopeFilters=lg%3Aroot%2Ctrue%3Br%3Aroot%2Ctrue%3Bg%3Aap-tokyo-1&timeNum=15&timeUnit=MINUTES®ion=ap-tokyo-1&tenant=xxxxxxxxxxx>

このURLは、後続の作業で利用するので、メモ帳またはテキストエディタにペーストして保存しておいてください。

以上で、問い合わせURLの取得は完了です。

OCI APMのドリルダウン設定

ナビゲーションメニューから「監視および管理」を選択して、「管理」をクリックします (図-75)。



図 -75 OCI APMのドリルダウン設定 1

「mushop」のテキストリンクをクリックします (図-76)。



図 -76 OCI APMのドリルダウン設定 2

画面右にある「リソース」メニューからの「ドリルダウン構成」のテキストリンクをクリックします (図-77)。

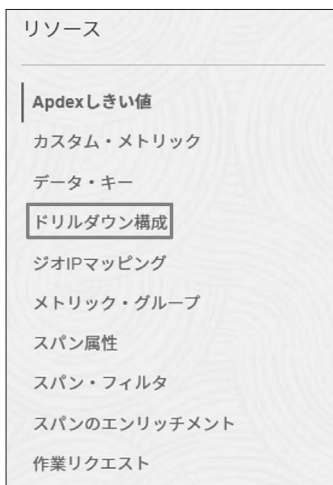


図-77 OCI APMのドリルダウン設定 3

「ドリルダウンの作成」ボタンをクリックします (図-78)。



図-78 OCI APMのドリルダウン設定 4

設定する内容は、表-28 の通りです。設定後、「保存」ボタンをクリックします (図-79)。

ドリルダウンの作成

ドリルダウンによって、現在のスパンの値で置き換えられたパラメータ値を使用して他のOCIアプリケーションに対するリンクを作成できます。ドリルダウンに条件を追加して、ドリルダウンを使用するスパンを制限できます。URLで'<'を使用して属性値を参照します。例:
http://cloud.oracle.com/pluginname?sqlId=<DbOracleSqlId>

名前

①

トレースとログの相関

説明

URL ①

②

https://cloud.oracle.com/loganalytics/explorer?viz=records&query=%27Log+Source%27+%3D+%27OCI+Unified+Schema+Logs%27+and+%27Trace+ID%27+%3D+<TraceId>&vizOptions=%27B%22customVizOpt%22%3A%7B%22primaryFieldName%22%3A%22mbody%22%22%22primaryFieldDname%22%3A%22Original%20Log%20Content%22%7D%7D&scopeFilters=lg%3Aroot%2Ctrue%3Brs%3Aroot%2Ctrue%3Brg%3Aap-tokyo-1&timeNum=15&timeUnit=MINUTES®ion=ap-tokyo-1&tenant=

☒ Enabled

条件

このドリルダウンでスパンを制限するオプションの条件。例: DbType = 'sq' および DbSystem = 'oracle'

③

保存

取消

図 -79 OCI APMのドリルダウン設定 5

設定項目	内容
名前	トレースとログの相関
URL	「問い合わせ URL の取得」で作成した URL https://cloud.oracle.com/loganalytics/explorer?viz=records&query=%27Log+Source%27+%3D+%27OCI+Unified+Schema+Logs%27+and+%27Trace+ID%27+%3D+<TraceId>&vizOptions=%27B%22customVizOpt%22%3A%7B%22primaryFieldName%22%3A%22mbody%22%22%22primaryFieldDname%22%3A%22Original%20Log%20Content%22%7D%7D&scopeFilters=lg%3Aroot%2Ctrue%3Brs%3Aroot%2Ctrue%3Brg%3Aap-tokyo-1&timeNum=15&timeUnit=MINUTES®ion=ap-tokyo-1&tenant=xxxxxxxxxx

表 -28 「ドリルダウンの作成」の設定値

「ドリルダウン構成」のリストに追加されます (図-80)。

ドリルダウン構成						
ドリルダウンの作成						
名前 URL 条件 説明 Created by Enabled						
Appserver Dashboard	apm/apm-traces/dashboards?dashid=OOBD-APM-app...	ApriVersion is not omitted	Drilldown to the APM Ap...	Oracle	✓	⋮
Autonomous Database (Template)	/db/adbi/replace with db ocid=	Component = 'JDBC'	Duplicate this drilldown a...	Oracle	✓	⋮
LogAn (Compute)	/loganalytics/explorer?viz=records_histogram&query=<...		Search for the OCI comp...	Oracle	✓	⋮
Opsinsights SQL Insights	/ops/dashboards/dashboards?dashid=OOBD-opsi-apm...		Drills into a dashboard s...	Oracle	✓	⋮
Perhub (Template)	dtmgmt-ui/perhub?ocid=<replace with db ocid=		Duplicate this drilldown a...	Oracle	✓	⋮
SQL Warehouse	opsi/sqi-warehouse/compartmentIdinSubtree=true&sql...		This URL will land in the ...	Oracle	✓	⋮
Traces in Session	apm/apm-traces/apmFilter/show%20(traces)%20%20...		See all the traces in the ...	Oracle	✓	⋮
トレースとログの関連	https://cloud.oracle.com/loganalytics/explorer?viz=recor...				✓	⋮

図-80 OCI APMのドリルダウン設定 6

以上で、OCI APMのドリルダウン設定は完了です。

≫ トレースIDをもとにしたログ分析の実践

トレースIDをもとにしたログ分析を実践してみましょう。ログが収集され、Logging Analyticsに反映されるまでは数分程度かかります。

最初に、MuShopで、商品をカートに入れ、注文し、配送するまでのシナリオを実施します。（「5-4-3. MuShop利用ガイダンス」の商品注文と配送処理）

次に、ナビゲーションメニューから「監視および管理」を選択して、「トレース・エクスプローラ」をクリックします（図-81）。

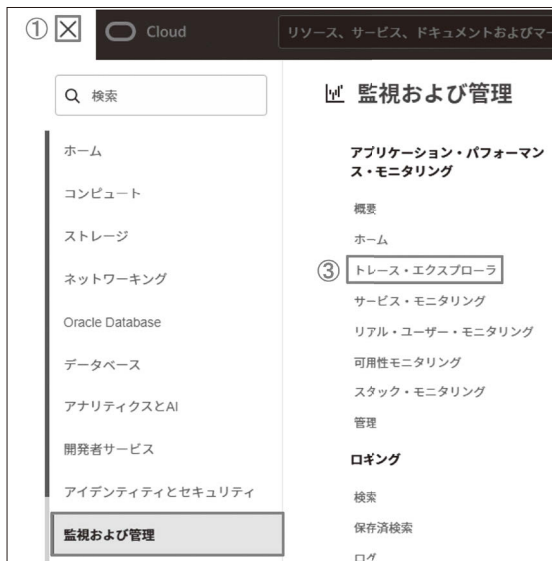


図-81 トレースIDをもとにしたログ分析の実践 1

トレース「Mushop-Service: Full Update /orders.html」のリンクテキストをクリックします (図-82)。

トレース 問合せWhere句に追加するグローバル・フィルタ							🔍 100を表示中 123 456
Service: Operation	Status	Start time	Duration	Spans	Span errors		
mushop-orders: GET /actuator/health**	●完了	18.06...	1ミリ秒	2	0		⋮
mushop-orders: GET /actuator/health**	●完了	18.06...	1ミリ秒	2	0		⋮
mushop-orders: GET /actuator/health**	●完了	18.06...	1ミリ秒	2	0		⋮
mushop-orders: GET /actuator/health**	●完了	18.06...	1ミリ秒	2	0		⋮
mushop-orders: GET /actuator/health**	●完了	18.06...	1ミリ秒	2	0		⋮
mushop-orders: GET /actuator/health**	●完了	18.06...	1ミリ秒	2	0		⋮
Mushop-Service: Full Update /orders.html	●成功	18.06...	11.18秒	59	0		⋮
Mushop-Service: Connection http://138.3.213.174	●完了	18.06...	12ミリ秒	1	0		⋮
Mushop-Service: Full Update /checkout.html	●成功	18.06...	207ミリ秒	5	0		⋮
mushop-orders: GET /actuator/health**	●完了	18.06...	1ミリ秒	2	0		⋮
Mushop-Service: Partial Update /subcategory.html	●成功	18.06...	295ミリ秒	14	0		⋮

図-82 トレースIDをもとにしたログ分析の実践 2

スパン「mushop-orders: POST /orders」のリンクテキストをクリックします (図-83)。

スパン				🔍 サービス、操作、またはコンポーネント名による検索 59を表示中
☐ エラーのみ	スパン	コンポーネント	0 32ミソ秒	10秒
	! Mushop-Service: Ajax /orders	BROWSER	307ミリ秒	⋮
	! Mushop-orders: POST /orders	—	113ミリ秒	⋮
	! Mushop-orders: OrdersController.newOrder	—	112ミリ秒	⋮
	! mushop-orders: GET	—	12ミリ秒	⋮

図-83 トレースIDをもとにしたログ分析の実践 3

トレースIDを確認して、「使用可能なドリルダウン」にある「トレースとログの相関」ボタンをクリックします (図-84)。

スパン詳細

サービス: mushop-orders

操作: POST /orders

種類: SERVER

起動済: 2025年5月14日(水) 18:06:14.676 JST

トレース開始後の時間: 253ミリ秒

スパンID: 03471dda085bd3b3

期間: 113ミリ秒

トレースID: b05b228dd45ded0d63c986ca5bcff138

使用可能なドリルダウン

ドリルダウンの管理

トレースとログの相関

図-84 トレースIDをもとにしたログ分析の実践 4

新規にブラウザウィンドウが起動して、トレース ID 「b05b228dd45ded0d63c986ca5bcff138」に紐づくログが表示されます (図-85)。

The screenshot shows the 'Log Explorer' application. The top bar indicates the log source is 'OCI Unified Schema Logs' and the trace ID is 'b05b228dd45ded0d63c986ca5bcff138'. The left sidebar contains a 'ビジュアライゼーション' (Visualization) section with various filter options like 'Entity', 'Host Name', 'Log Group', etc. The main area displays a table of log entries with columns for 'Time (UTC+09:00)' and 'Original Log Content'. The right-hand pane shows the 'Original Log Content' for the selected log entry, which includes details about the OCI Unified Schema Logs and the specific log entry.

図-85 トレースIDをもとにしたログ分析の実践 5

「ビジュアライゼーション」の表示パターンを切り替えて、様々なログの見え方を確認してみましょう (図-86)。

終了する場合は、ブラウザを閉じてください。

The screenshot shows the 'Log Explorer' application with the 'ビジュアライゼーション' (Visualization) section selected. The main view displays a table of log entries with columns for 'Time (UTC+09:00)' and 'Original Log Content'. The right-hand pane shows the 'Original Log Content' for the selected log entry, which includes details about the OCI Unified Schema Logs and the specific log entry. The interface also includes a 'ヒストグラム付きレコード' (Record with Histogram) section and a 'ビジュアライゼーション' (Visualization) section with various filter options.

図-86 トレースIDをもとにしたログ分析の実践 6

最後に「スパンの詳細」と「トレースの詳細」の「閉じる」ボタンをクリックして終了します(図-87)。

The screenshot displays the Oracle Cloud console interface for analyzing a trace. The left sidebar shows a tree view with 'Mushop-Service: Full Update /orders.html' selected. The main area is divided into two panels: 'Trace Details' (left) and 'Span Details' (right).

Trace Details (Left Panel):

- Trace ID:** 05471d5a0850c3b3
- Span ID:** b55b229d450ed3b3c596ca5bc1138
- Service:** mushop-orders
- Operation:** POST /orders
- Status:** 成功 (Success)
- Span Error:** 0
- Trace Start Time:** 2025年5月14日(水) 18:06:14.676 JST
- Trace End Time:** 2025年5月14日(水) 18:06:14.789 JST
- Trace Duration:** 253.5 ミリ秒

Span Details (Right Panel):

- Span ID:** 56
- Parent Span ID:** 56
- Attributes:**
 - ApdexLevel:** satisfied
 - ApdexScore:** 1
 - container.id:** 09e3870a3aa8f73bfb02561a433c23512ea249680c4d3147325bfe15a7679c
 - EndTime:** 2025年5月14日(水) 18:06:14.789 JST
 - host.arch:** amd64
 - host.name:** mushop-orders-76d4b6c59k-mduah
 - http.method:** POST
 - http.request.content.length:** 392
 - http.route:** /orders
 - http.scheme:** http
 - http.status_code:** 201

At the bottom of the span details panel, there are buttons for '閉じる' (Close) and 'スパンの比較' (Compare Spans).

図-87 トレースIDをもとにしたログ分析の実践 7

以上で、トレースIDをもとにしたログ分析の実践は完了です。

参考情報

日本オラクル株式会社では、クラウドネイティブ技術を中心としたミートアップイベントを定期的に開催しています(執筆時点)。

Oracle Cloud Hangout Cafe

Oracle Cloud Hangout Cafe、略してOCHa Cafe (おちゃかふえ) は、日本オラクル株式会社が提供する、クラウドネイティブ時代の開発者やITプロフェッショナルが集まるコミュニティ型のテクニカル勉強会シリーズです。ハッシュタグは、「#ochacafe」です。

業界で注目されるオープンスタンダードな技術をテーマに、短時間で深く学び、実践で役立

つ知識やスキルを習得できる場を提供しています。このコミュニティを通じて、参加者同士のネットワーキングや情報交換の機会も広がります。

さらに、スピンオフ企画として「OCHa Cafe Premium」も不定期で開催しています。通常版がオープンスタンダード技術を扱うのに対し、プレミアムではOracleの技術に特化し、実践的なノウハウや最新情報を提供します。Oracleならではの専門性と実績を活かした内容で、現場に直結する成果を目指していただけます（図-88）。

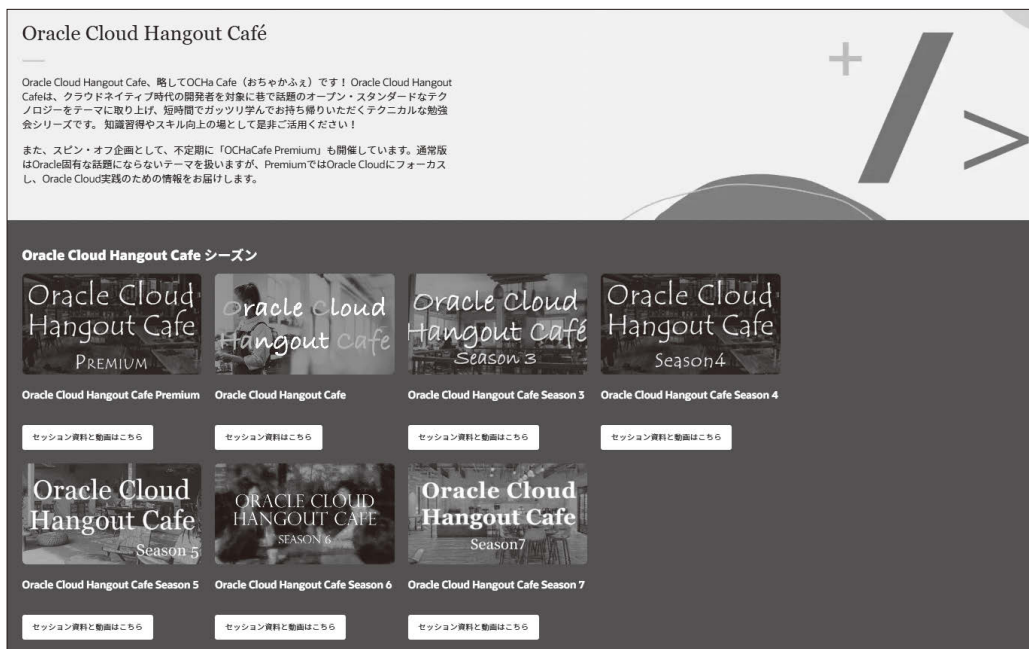


図-88 Oracle Cloud Hangout Café

コミュニティサイトおよびこれまでの動画と資料は、以下のサイトから閲覧できます。

- コミュニティサイト：<https://ochacafe.connpass.com/>
- アーカイブサイト：<https://www.oracle.com/jp/developer/events/cloud-hangout-cafe/>

これまで開催してきた中で、ご好評いただいたテーマをダイジェストとして記事を公開しています。

- 「Oracle Cloud Hangout Cafe (OCHaCafe)」ダイジェスト 1 : <https://thinkit.co.jp/series/10728>
- 「Oracle Cloud Hangout Cafe (OCHaCafe)」ダイジェスト 2 : <https://thinkit.co.jp/series/11131>