OpenShift Container Platform 4.7

ノード

OpenShift Container Platform でのノードの設定および管理

Last Updated: 2023-02-24

OpenShift Container Platform 4.7 ノード

OpenShift Container Platform でのノードの設定および管理

法律上の通知

概要

本書では、クラスターのノード、Pod、コンテナーを設定し管理する方法について説明します。また、Pod のスケジューリングや配置の設定方法、ジョブや DeamonSet を使用してタスクを自動化する方法やクラスターを効率化するための他のタスクなどに関する情報も提供します。

目次

第1章 ノードの概要 **4**
1.1. ノードについて 4
1.2. POD について 5
1.3. コンテナーについて 6

第2章 POD の使用 **8**
2.1. POD の使用 8
2.2. POD の表示 11
2.3. OPENSHIFT CONTAINER PLATFORM クラスターでの POD の設定 14
2.4. HORIZONTAL POD AUTOSCALER での POD の自動スケーリング 18
2.5. VERTICAL POD AUTOSCALER を使用した POD リソースレベルの自動調整 35
2.6. POD への機密性の高いデータの提供 47
2.7. 設定マップの作成および使用 60
2.8. POD で外部リソースにアクセスするためのデバイスプラグインの使用 71
2.9. POD スケジューリングの決定に POD の優先順位を含める 74
2.10. ノードセレクターの使用による特定ノードへの POD の配置 78

第3章 POD のノードへの配置の制御 (スケジューリング) **82**
3.1. スケジューラーによる POD 配置の制御 82
3.2. デフォルトスケジューラーの設定による POD 配置の制御 83
3.3. スケジューラープロファイルを使用した POD のスケジューリング 99
3.4. アフィニティールールと非アフィニティールールの使用による他の POD との相対での POD の配置 101
3.5. ノードのアフィニティールールを使用したノード上での POD 配置の制御 108
3.6. POD のオーバーコミットノードへの配置 113
3.7. ノードテイントを使用した POD 配置の制御 115
3.8. ノードセレクターの使用による特定ノードへの POD の配置 128
3.9. POD トポロジー分散制約を使用した POD 配置の制御 140
3.10. カスタムスケジューラーの実行 143
3.11. DESCHEDULER を使用した POD のエビクト 148

第4章 ジョブと DEAMONSET の使用 **155**
4.1. デーモンセットによるノード上でのバックグラウンドタスクの自動的な実行 155
4.2. ジョブの使用による POD でのタスクの実行 158

第5章 ノードの使用 **165**
5.1. OPENSHIFT CONTAINER PLATFORM クラスター内のノードの閲覧と一覧表示 165
5.2. ノードの使用 171
5.3. ノードの管理 180
5.4. ノードあたりの POD の最大数の管理 182
5.5. NODE TUNING OPERATOR の使用 184
5.6. ノードの再起動について 192
5.7. ガベージコレクションを使用しているノードリソースの解放 195
5.8. OPENSHIFT CONTAINER PLATFORM クラスター内のノードのリソースの割り当て 200
5.9. クラスター内のノードの特定 CPU の割り当て 203
5.10. MACHINE CONFIG DAEMON メトリクス 204

第6章 コンテナーの使用 **208**
6.1. コンテナーについて 208
6.2. POD のデプロイ前の、INIT コンテナーの使用によるタスクの実行 208
6.3. ボリュームの使用によるコンテナーデータの永続化 211
6.4. PROJECTED ボリュームによるボリュームのマッピング 218
6.5. コンテナーによる API オブジェクト使用の許可 225

6.6. OPENSHIFT CONTAINER PLATFORM コンテナーへの/からのファイルのコピー 233
6.7. OPENSHIFT CONTAINER PLATFORM コンテナーでのリモートコマンドの実行 235
6.8. コンテナー内のアプリケーションにアクセスするためのポート転送の使用 237
6.9. コンテナーでの SYSCTL の使用 239

第7章 クラスターの操作 245
7.1. OPENSHIFT CONTAINER PLATFORM クラスター内のシステムイベント情報の表示 245
7.2. OPENSHIFT CONTAINER PLATFORM のノードが保持できる POD の数の見積り 254
7.3. 制限範囲によるリソース消費の制限 259
7.4. コンテナーメモリーとリスク要件を満たすためのクラスターメモリーの設定 267
7.5. オーバーコミットされたノード上に POD を配置するためのクラスターの設定 274
7.6. FEATUREGATE の使用による OPENSHIFT CONTAINER PLATFORM 機能の有効化 288

第8章 ネットワークエッジ上にあるリモートワーカーノード 292
8.1. ネットワークエッジでのリモートワーカーノードの使用 292

第1章 ノードの概要

1.1. ノードについて

ノードは、Kubernetes クラスター内の仮想マシンまたはベアメタルマシンです。ワーカーノードは、Pod としてグループ化されたアプリケーションコンテナーをホストします。コントロールプレーンノードは、Kubernetes クラスターを制御するために必要なサービスを実行します。OpenShift Container Platform では、コントロールプレーンノードには、OpenShift ContainerPlatform クラスターを管理するための Kubernetes サービス以上のものが含まれています。

クラスター内に安定した正常なノードを持つことは、ホストされたアプリケーションがスムーズに機能するための基本です。OpenShift Container Platform では、ノードを表す Node オブジェクトを介して **Node** にアクセス、管理、およびモニターできます。OpenShift CLI (**oc**) または Web コンソールを使用して、ノードで以下の操作を実行できます。

読み取り操作
読み取り操作により、管理者または開発者は OpenShift ContainerPlatform クラスター内のノードに関する情報を取得できます。

- クラスター内のすべてのノードを一覧表示します。
- メモリーと CPU の使用率、ヘルス、ステータス、経過時間など、ノードに関する情報を取得します。
- ノードで実行されている Pod を一覧表示します。

管理操作
管理者は、次のいくつかのタスクを通じて、OpenShift ContainerPlatform クラスター内のノードを簡単に管理できます。

- ノードラベルを追加または更新します。ラベルは、**Node** オブジェクトに適用されるキーと値のペアです。ラベルを使用して Pod のスケジュールを制御できます。
- カスタムリソース定義 (CRD) または **kubeletConfig** オブジェクトを使用してノード設定を変更します。
- Pod のスケジューリングを許可または禁止するようにノードを設定します。ステータスが **Ready** の正常なワーカーノードでは、デフォルトで Pod の配置が許可されますが、コントロールプレーンノードでは許可されません。このデフォルトの動作を変更するには、ワーカーノードをスケジュール不可に設定 し、コントロールプレーンノードをスケジュール可能に設定 します。
- **system-reserved** 設定を使用して、ノードにリソースを割り当てます。OpenShift Container Platform がノードに最適な **system-reserved**CPU およびメモリーリソースを自動的に決定できるようにするか、ノードに最適なリソースを手動で決定および設定することができます。
- ノード上のプロセッサーコアの数、ハード制限、またはその両方に基づいて、ノード上で実行できる Pod の数を設定 します。
- Pod の非アフィニティー を使用して、ノードを正常に再起動します。
- マシンセットを使用してクラスターをスケールダウンすることにより、クラスターからノードを削除 します。ベアメタルクラスターからノードを削除するには、最初にノード上のすべての Pod をドレインしてから、手動でノードを削除する必要があります。

エンハンスメント操作

OpenShift Container Platform を使用すると、ノードへのアクセスと管理以上のことができます。管理者は、ノードで次のタスクを実行して、クラスターをより効率的でアプリケーションに適したものにし、開発者により良い環境を提供できます。

- Node Tuning Operator を使用 して、ある程度のカーネルチューニングを必要とする高性能アプリケーションのノードレベルのチューニングを管理します。
- デーモンセットを使用してノードでバックグラウンドタスクを自動的に実行 します。デーモンセットを作成して使用し、共有ストレージを作成したり、すべてのノードでロギング Pod を実行したり、すべてのノードに監視エージェントをデプロイしたりできます。
- ガベージコレクションを使用してノードリソースを解放 します。終了したコンテナーと、実行中の Pod によって参照されていないイメージを削除することで、ノードが効率的に実行されていることを確認できます。
- カーネル引数をノードのセットに追加 します。
- ネットワークエッジにワーカーノード (リモートワーカーノード) を持つように OpenShift ContainerPlatform クラスターを設定します。OpenShift Container Platform クラスターにリモートワーカーノードを配置する際の課題と、リモートワーカーノードで Pod を管理するための推奨されるアプローチについては、ネットワークエッジでのリモートワーカーノードの使用 を参照してください。

1.2. POD について

Pod は、ノードに一緒にデプロイされる 1 つ以上のコンテナーです。クラスター管理者は、Pod を定義し、スケジューリングの準備ができている正常なノードで実行するように割り当て、管理することができます。コンテナーが実行されている限り、Pod は実行されます。Pod を定義して実行すると、Pod を変更することはできません。Pod を操作するときに実行できる操作は次のとおりです。

読み取り操作
管理者は、次のタスクを通じてプロジェクト内の Pod に関する情報を取得できます。

- レプリカと再起動の数、現在のステータス、経過時間などの情報を含む、プロジェクトに関連付けられている Pod を一覧表示 します。
- CPU、メモリー、ストレージ消費量などの Pod 使用統計を表示 します。

管理操作
以下のタスクのリストは、管理者が OpenShift ContainerPlatform クラスターで Pod を管理する方法の概要を示しています。

- OpenShift Container Platform で利用可能な高度なスケジューリング機能を使用して、Pod のスケジューリングを制御します。
  - Pod アフィニティー、ノードアフィニティー、非アフィニティー などのノード間バインディングルール。
  - ノードラベルとセレクター。
  - テイントおよび容認 (Toleration)。
  - Pod トポロジー分散制約。
  - カスタムスケジューラー。

- 特定のストラテジーに基づいて Pod をエビクトするように descheduler を設定 して、スケジューラーが Pod をより適切なノードに再スケジュールするようにします。
- Pod コントローラーと再起動ポリシーを使用して、再起動後の Pod の動作を設定します。
- Pod の egress トラフィック と ingress トラフィックの両方を制限 します。
- Pod テンプレートを持つオブジェクトとの間でボリュームを追加および削除 します。ボリュームは、Pod 内のすべてのコンテナーで使用できるマウントされたファイルシステムです。コンテナーの保管はエフェメラルなものです。ボリュームを使用して、コンテナーデータを永続化できます。

エンハンスメント操作

OpenShift Container Platform で利用可能なさまざまなツールと機能を使用して、Pod をより簡単かつ効率的に操作できます。次の操作では、これらのツールと機能を使用して Pod をより適切に管理します。

操作	ユーザー	詳細情報
Horizontal Pod Autoscaler を作成して使用。	開発者	Horizontal Pod Autoscaler を使用して、実行する Pod の最小数と最大数、および Pod がターゲットとする CPU 使用率またはメモリー使用率を指定できます。水平 Pod オートスケーラーを使用すると、Pod を 自動的にスケーリング できます。
垂直 Pod オートスケーラーをインストールして使用。	管理者および開発者	管理者は、垂直 Pod オートスケーラーを使用して、リソースとワークロードのリソース要件を監視することにより、クラスターリソースをより適切に使用します。 開発者は、垂直 Pod オートスケーラーを使用して、各 Pod に十分なリソースがあるノードに Pod をスケジュールすることにより、需要が高い時に Pod が稼働し続けるようにします。
デバイスプラグインを使用して外部リソースへのアクセスを提供します。	Administrator	デバイスプラグイン は、ノード (kubelet の外部) で実行される gRPC サービスであり、特定のハードウェアリソースを管理します。デバイスプラグインを導入 して、クラスター全体でハードウェアデバイスを使用するための一貫性のあるポータブルソリューションを提供できます。
Secret オブジェクトを使用 して機密データを Pod に提供。	Administrator	一部のアプリケーションでは、パスワードやユーザー名などの機密情報が必要です。**Secret** オブジェクトを使用して、そのような情報をアプリケーション Pod に提供できます。

1.3. コンテナーについて

コンテナーは、OpenShift Container Platform アプリケーションの基本ユニットであり、依存関係、ライブラリー、およびバイナリーとともにパッケージ化されたアプリケーションコードで設定されます。コンテナーは、複数の環境、および物理サーバー、仮想マシン (VM)、およびプライベートまたはパブリッククラウドなどの複数のデプロイメントターゲット間に一貫性をもたらします。

Linux コンテナーテクノロジーは、実行中のプロセスを分離し、指定されたリソースのみへのアクセスを制限するための軽量メカニズムです。管理者は、Linux コンテナーで次のようなさまざまなタスクを実行できます。

- コンテナーとの間でファイルをコピー します。
- コンテナーによる API オブジェクトの消費を許可 します。
- コンテナー内でリモートコマンドを実行 します。
- ポート転送を使用してコンテナー内のアプリケーションにアクセス します。

OpenShift Container Platform は、Init コンテナー と呼ばれる特殊なコンテナーを提供します。Init コンテナーは、アプリケーションコンテナーの前に実行され、アプリケーションイメージに存在しないユーティリティーまたはセットアップスクリプトを含めることができます。Pod の残りの部分がデプロイされる前に、Init コンテナーを使用してタスクを実行できます。

ノード、Pod、およびコンテナーで特定のタスクを実行する以外に、OpenShift Container Platform クラスター全体を操作して、クラスターの効率とアプリケーション Pod の高可用性を維持できます。

第2章 POD の使用

2.1. POD の使用

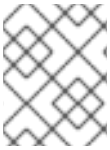
Pod は 1 つのホストにデプロイされる 1 つ以上のコンテナーであり、定義され、デプロイされ、管理される最小のコンピュート単位です。

2.1.1. Pod について

Pod はコンテナーに対してマシンインスタンス (物理または仮想) とほぼ同じ機能を持ちます。各 Pod は独自の内部 IP アドレスで割り当てられるため、そのポートスペース全体を所有し、Pod 内のコンテナーはそれらのローカルストレージおよびネットワークを共有できます。

Pod にはライフサイクルがあります。それらは定義された後にノードで実行されるために割り当てられ、コンテナーが終了するまで実行されるか、その他の理由でコンテナーが削除されるまで実行されます。ポリシーおよび終了コードによっては、Pod は終了後に削除されるか、コンテナーのログへのアクセスを有効にするために保持される可能性があります。

OpenShift Container Platform は Pod をほとんどがイミュータブルなものとして処理します。Pod が実行中の場合は Pod に変更を加えることができません。OpenShift Container Platform は既存 Pod を終了し、これを変更された設定、ベースイメージのいずれかまたはその両方で再作成して変更を実装します。Pod は拡張可能なものとしても処理されますが、再作成時に状態を維持しません。そのため、通常 Pod はユーザーから直接管理されるのでははく、ハイレベルのコントローラーで管理される必要があります。



注記

OpenShift Container Platform ノードホストごとの Pod の最大数については、クラスターの制限について参照してください。

警告

レプリケーションコントローラーによって管理されないベア Pod はノードの中断時に再スケジュールされません。

2.1.2. Pod 設定の例

OpenShift Container Platform は、**Pod** の Kubernetes の概念を活用しています。これはホスト上に共にデプロイされる 1 つ以上のコンテナーであり、定義され、デプロイされ、管理される最小のコンピュート単位です。

以下は、Rails アプリケーションからの Pod の定義例です。これは数多くの Pod の機能を示していますが、それらのほとんどは他のトピックで説明されるため、ここではこれらについて簡単に説明します。

Pod オブジェクト定義 (YAML)

```
kind: Pod
apiVersion: v1
metadata:
```

```
  name: example
  namespace: default
  selfLink: /api/v1/namespaces/default/pods/example
  uid: 5cc30063-0265780783bc
  resourceVersion: '165032'
  creationTimestamp: '2019-02-13T20:31:37Z'
  labels:
    app: hello-openshift (1)
  annotations:
    openshift.io/scc: anyuid
spec:
  restartPolicy: Always (2)
  serviceAccountName: default
  imagePullSecrets:
    - name: default-dockercfg-5zrhb
  priority: 0
  schedulerName: default-scheduler
  terminationGracePeriodSeconds: 30
  nodeName: ip-10-0-140-16.us-east-2.compute.internal
  securityContext: (3)
    seLinuxOptions:
      level: 's0:c11,c10'
  containers: (4)
    - resources: {}
      terminationMessagePath: /dev/termination-log
      name: hello-openshift
      securityContext:
        capabilities:
          drop:
            - MKNOD
        procMount: Default
      ports:
        - containerPort: 8080
          protocol: TCP
      imagePullPolicy: Always
      volumeMounts: (5)
        - name: default-token-wbqsl
          readOnly: true
          mountPath: /var/run/secrets/kubernetes.io/serviceaccount (6)
      terminationMessagePolicy: File
      image: registry.redhat.io/openshift4/ose-ogging-eventrouter:v4.3 (7)
  serviceAccount: default (8)
  volumes: (9)
    - name: default-token-wbqsl
      secret:
        secretName: default-token-wbqsl
        defaultMode: 420
  dnsPolicy: ClusterFirst
status:
  phase: Pending
  conditions:
    - type: Initialized
      status: 'True'
      lastProbeTime: null
      lastTransitionTime: '2019-02-13T20:31:37Z'
```

```
  - type: Ready
    status: 'False'
    lastProbeTime: null
    lastTransitionTime: '2019-02-13T20:31:37Z'
    reason: ContainersNotReady
    message: 'containers with unready status: [hello-openshift]'
  - type: ContainersReady
    status: 'False'
    lastProbeTime: null
    lastTransitionTime: '2019-02-13T20:31:37Z'
    reason: ContainersNotReady
    message: 'containers with unready status: [hello-openshift]'
  - type: PodScheduled
    status: 'True'
    lastProbeTime: null
    lastTransitionTime: '2019-02-13T20:31:37Z'
hostIP: 10.0.140.16
startTime: '2019-02-13T20:31:37Z'
containerStatuses:
  - name: hello-openshift
    state:
      waiting:
        reason: ContainerCreating
    lastState: {}
    ready: false
    restartCount: 0
    image: openshift/hello-openshift
    imageID: ''
qosClass: BestEffort
```

1. Pod には 1 つまたは複数のラベルでタグ付けすることができ、このラベルを使用すると、一度の操作で Pod グループの選択や管理が可能になります。これらのラベルは、キー/値形式で **metadata** ハッシュに保存されます。

2. Pod 再起動ポリシーと使用可能な値の **Always**、**OnFailure**、および **Never** です。デフォルト値は **Always** です。

3. OpenShift Container Platform は、コンテナーが特権付きコンテナーとして実行されるか、選択したユーザーとして実行されるかどうかを指定するセキュリティーコンテキストを定義します。デフォルトのコンテキストには多くの制限がありますが、管理者は必要に応じてこれを変更できます。

4. **containers** は、1 つ以上のコンテナー定義の配列を指定します。

5. コンテナーは外部ストレージボリュームがコンテナー内にマウントされるかどうかを指定します。この場合、OpenShift Container Platform API に対して要求を行うためにレジストリーが必要とする認証情報へのアクセスを保存するためにボリュームがあります。

6. Pod に提供するボリュームを指定します。ボリュームは指定されたパスにマウントされます。コンテナーのルート (/) や、ホストとコンテナーで同じパスにはマウントしないでください。これは、コンテナーに十分な特権が付与されている場合、ホストシステムを破壊する可能性があります (例: ホストの **/dev/pts** ファイル)。ホストをマウントするには、**/host** を使用するのが安全です。

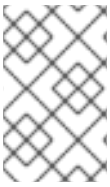
7. Pod 内の各コンテナーは、独自のコンテナーイメージからインスタンス化されます。

8. OpenShift Container Platform API に対して要求する Pod は一般的なパターンです。この場合、**serviceAccount** フィールドがあり、これは要求を行う際に Pod が認証する必要のあるサー

ビスアカウントユーザーを指定するために使用されます。これにより、カスタムインフラストラクチャーコンポーネントの詳細なアクセス制御が可能になります。

9 Pod は、コンテナーで使用できるストレージボリュームを定義します。この場合、デフォルトのサービスアカウントトークンを含む **secret** ボリュームのエフェメラルボリュームを提供します。

ファイル数が多い永続ボリュームを Pod に割り当てる場合、それらの Pod は失敗するか、または起動に時間がかかる場合があります。詳細は、When using Persistent Volumes with high file counts in OpenShift, why do pods fail to start or take an excessive amount of time to achieve "Ready" state? を参照してください。



注記

この Pod 定義には、Pod が作成され、ライフサイクルが開始された後に OpenShift Container Platform によって自動的に設定される属性が含まれません。Kubernetes Pod ドキュメント には、Pod の機能および目的についての詳細が記載されています。

2.1.3. 関連情報

- Pod とストレージの詳細については、Understanding persistent storage と Understanding ephemeral storage を参照してください。

2.2. POD の表示

管理者として、クラスターで Pod を表示し、それらの Pod および全体としてクラスターの正常性を判別することができます。

2.2.1. Pod について

OpenShift Container Platform は、**Pod** の Kubernetes の概念を活用しています。これはホスト上に共にデプロイされる 1 つ以上のコンテナーであり、定義され、デプロイされ、管理される最小のコンピュート単位です。Pod はコンテナーに対するマシンインスタンス (物理または仮想) とほぼ同等のものです。

特定のプロジェクトに関連付けられた Pod の一覧を表示したり、Pod についての使用状況の統計を表示したりすることができます。

2.2.2. プロジェクトでの Pod の表示

レプリカの数、Pod の現在のステータス、再起動の数および年数を含む、現在のプロジェクトに関連付けられた Pod の一覧を表示できます。

手順

プロジェクトで Pod を表示するには、以下を実行します。

1. プロジェクトに切り替えます。

```
$ oc project <project-name>
```

2. 以下のコマンドを実行します。

```
$ oc get pods
```

以下に例を示します。

```
$ oc get pods -n openshift-console
```

出力例

```
NAME                  READY   STATUS    RESTARTS   AGE
console-698d866b78-bnshf   1/1     Running   2          165m
console-698d866b78-m87pm   1/1     Running   2          165m
```

-o wide フラグを追加して、Pod の IP アドレスと Pod があるノードを表示します。

```
$ oc get pods -o wide
```

出力例

```
NAME                  READY   STATUS    RESTARTS   AGE    IP           NODE
NOMINATED NODE
console-698d866b78-bnshf   1/1     Running   2          166m   10.128.0.24   ip-10-0-152-
71.ec2.internal    <none>
console-698d866b78-m87pm   1/1     Running   2          166m   10.129.0.23   ip-10-0-173-
237.ec2.internal   <none>
```

2.2.3. Pod の使用状況についての統計の表示

コンテナーのランタイム環境を提供する、Pod についての使用状況の統計を表示できます。これらの使用状況の統計には CPU、メモリー、およびストレージの消費量が含まれます。

前提条件

- 使用状況の統計を表示するには、**cluster-reader** パーミッションがなければなりません。
- 使用状況の統計を表示するには、メトリクスをインストールしている必要があります。

手順

使用状況の統計を表示するには、以下を実行します。

1. 以下のコマンドを実行します。

```
$ oc adm top pods
```

以下に例を示します。

```
$ oc adm top pods -n openshift-console
```

出力例

```
NAME                         CPU(cores)   MEMORY(bytes)
console-7f58c69899-q8c8k     0m           22Mi
console-7f58c69899-xhbgg     0m           25Mi
```

```
downloads-594fcccf94-bcxk8   3m          18Mi
downloads-594fcccf94-kv4p6   2m          15Mi
```

2. ラベルを持つ Pod の使用状況の統計を表示するには、以下のコマンドを実行します。

```
$ oc adm top pod --selector=''
```

フィルターに使用するセレクター (ラベルクエリー) を選択する必要があります。**=**、**==**、および **!=** をサポートします。

2.2.4. リソースログの表示

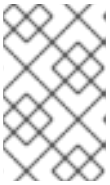
OpenShift CLI (oc) および Web コンソールで、各種リソースのログを表示できます。ログの末尾から読み取られるログ。

前提条件

- OpenShift CLI (oc) へのアクセス。

手順 (UI)

1. OpenShift Container Platform コンソールで **Workloads → Pods** に移動するか、または調査するリソースから Pod に移動します。



注記

ビルドなどの一部のリソースには、直接クエリーする Pod がありません。このような場合には、リソースについて **Details** ページで **Logs** リンクを特定できます。

2. ドロップダウンメニューからプロジェクトを選択します。
3. 調査する Pod の名前をクリックします。
4. **Logs** をクリックします。

手順 (CLI)

- 特定の Pod のログを表示します。

```
$ oc logs -f <pod_name> -c <container_name>
```

ここでは、以下のようになります。

-f
オプション: ログに書き込まれている内容に沿って出力することを指定します。

<pod_name>
Pod の名前を指定します。

<container_name>
オプション: コンテナーの名前を指定します。Pod に複数のコンテナーがある場合、コンテナー名を指定する必要があります。

以下に例を示します。

```
$ oc logs ruby-58cd97df55-mww7r
```

```
$ oc logs -f ruby-57f7f4855b-znl92 -c ruby
```

ログファイルの内容が出力されます。

- 特定のリソースのログを表示します。

```
$ oc logs <object_type>/<resource_name> 1
```

1 リソースタイプおよび名前を指定します。

以下に例を示します。

```
$ oc logs deployment/ruby
```

ログファイルの内容が出力されます。

2.3. OPENSHIFT CONTAINER PLATFORM クラスターでの POD の設定

管理者として、Pod に対して効率的なクラスターを作成し、維持することができます。

クラスターの効率性を維持することにより、1 回のみ実行するように設計された Pod をいつ再起動するか、Pod が利用できる帯域幅をいつ制限するか、中断時に Pod をどのように実行させ続けるかなど、Pod が終了するときの動作をツールとして使って必要な数の Pod が常に実行されるようにし、開発者により良い環境を提供することができます。

2.3.1. 再起動後の Pod の動作方法の設定

Pod 再起動ポリシーは、Pod のコンテナーの終了時に OpenShift Container Platform が応答する方法を決定します。このポリシーは Pod のすべてのコンテナーに適用されます。

以下の値を使用できます。

- **Always**: Pod が再起動するまで、Pod で正常に終了したコンテナーの継続的な再起動を、指数関数のバックオフ遅延 (10 秒、20 秒、40 秒) で試行します。デフォルトは **Always** です。
- **OnFailure**: Pod で失敗したコンテナーの継続的な再起動を、5 分を上限として指数関数のバックオフ遅延 (10 秒、20 秒、40 秒) で試行します。
- **Never**: Pod で終了したコンテナーまたは失敗したコンテナーの再起動を試行しません。Pod はただちに失敗し、終了します。

いったんノードにバインドされた Pod は別のノードにはバインドされなくなります。これは、Pod がのノードの失敗後も存続するにはコントローラーが必要であることを示しています。

条件	コントローラーのタイプ	再起動ポリシー
(バッチ計算など) 終了することが予想される Pod	ジョブ	**OnFailure** または **Never**

条件	コントローラーのタイプ	再起動ポリシー
(Web サービスなど) 終了しないことが予想される Pod	レプリケーションコントローラー	**Always**
マシンごとに 1 回実行される Pod	デーモンセット	すべて

Pod のコンテナーが失敗し、再起動ポリシーが **OnFailure** に設定される場合、Pod はノード上に留まり、コンテナーが再起動します。コンテナーを再起動させない場合には、再起動ポリシーの **Never** を使用します。

Pod 全体が失敗すると、OpenShift Container Platform は新規 Pod を起動します。開発者は、アプリケーションが新規 Pod で再起動される可能性に対応しなくてはなりません。とくに、アプリケーションは、一時的なファイル、ロック、以前の実行で生じた未完成の出力などを処理する必要があります。

注記

Kubernetes アーキテクチャーでは、クラウドプロバイダーからの信頼性のあるエンドポイントが必要です。クラウドプロバイダーが停止している場合、kubelet は OpenShift Container Platform が再起動されないようにします。

基礎となるクラウドプロバイダーのエンドポイントに信頼性がない場合は、クラウドプロバイダー統合を使用してクラスターをインストールしないでください。クラスターを、非クラウド環境で実行する場合のようにインストールします。インストール済みのクラスターで、クラウドプロバイダー統合をオンまたはオフに切り替えることは推奨されていません。

OpenShift Container Platform が失敗したコンテナーについて再起動ポリシーを使用する方法の詳細は、Kubernetes ドキュメントの State の例 を参照してください。

2.3.2. Pod で利用可能な帯域幅の制限

QoS (Quality-of-Service) トラフィックシェーピングを Pod に適用し、その利用可能な帯域幅を効果的に制限することができます。(Pod からの) Egress トラフィックは、設定したレートを超えるパケットを単純にドロップするポリシングによって処理されます。(Pod への) Ingress トラフィックは、データを効果的に処理できるようシェーピングでパケットをキューに入れて処理されます。Pod に設定する制限は、他の Pod の帯域幅には影響を与えません。

手順

Pod の帯域幅を制限するには、以下を実行します。

1. オブジェクト定義 JSON ファイルを作成し、**kubernetes.io/ingress-bandwidth** および **kubernetes.io/egress-bandwidth** アノテーションを使用してデータトラフィックの速度を指定します。たとえば、 Pod の egress および ingress の両方の帯域幅を 10M/s に制限するには、以下を実行します。

   制限が設定された Pod オブジェクト定義

```
{
    "kind": "Pod",
    "spec": {
        "containers": [
```

```
            {
                "image": "openshift/hello-openshift",
                "name": "hello-openshift"
            }
        ]
    },
    "apiVersion": "v1",
    "metadata": {
        "name": "iperf-slow",
        "annotations": {
            "kubernetes.io/ingress-bandwidth": "10M",
            "kubernetes.io/egress-bandwidth": "10M"
        }
    }
}
```

2. オブジェクト定義を使用して Pod を作成します。

```
$ oc create -f <file_or_dir_path>
```

2.3.3. Pod の Disruption Budget (停止状態の予算) を使って起動している Pod の数を指定する方法

Pod の Disruption Budget は Kubernetes API の一部であり、他のオブジェクトタイプのように **oc** コマンドで管理できます。この設定により、メンテナンスのためのノードのドレイン (解放) などの操作時に Pod への安全面の各種の制約を指定できます。

PodDisruptionBudget は、同時に起動している必要のあるレプリカの最小数またはパーセンテージを指定する API オブジェクトです。これらをプロジェクトに設定することは、ノードのメンテナンス (クラスターのスケールダウンまたはクラスターのアップグレードなどの実行) 時に役立ち、この設定は (ノードの障害時ではなく) 自発的なエビクションの場合にのみ許可されます。

PodDisruptionBudget オブジェクトの設定は、以下の主要な部分で設定されています。

- 一連の Pod に対するラベルのクエリー機能であるラベルセレクター。
- 同時に利用可能にする必要のある Pod の最小数を指定する可用性レベル。
  - **minAvailable** は、中断時にも常に利用可能である必要のある Pod 数です。
  - **maxUnavailable** は、中断時に利用不可にできる Pod 数です。

注記

maxUnavailable の **0%** または **0** あるいは **minAvailable** の **100%**、ないしはレプリカ数に等しい値は許可されますが、これによりノードがドレイン (解放) されないようにブロックされる可能性があります。

以下を実行して、Pod の Disruption Budget をすべてのプロジェクトで確認することができます。

```
$ oc get poddisruptionbudget --all-namespaces
```

出力例

```
NAMESPACE        NAME         MIN-AVAILABLE   SELECTOR
another-project  another-pdb  4               bar=foo
test-project     my-pdb       2               foo=bar
```

PodDisruptionBudget は、最低でも **minAvailable** Pod がシステムで実行されている場合は正常であるとみなされます。この制限を超えるすべての Pod はエビクションの対象となります。

注記

Pod の優先順位およびプリエンプションの設定に基づいて、優先順位の低い Pod は Pod の Disruption Budget の要件を無視して削除される可能性があります。

2.3.3.1. Pod の Disruption Budget を使って起動している Pod 数の指定

同時に起動している必要のあるレプリカの最小数またはパーセンテージは、**PodDisruptionBudget** オブジェクトを使って指定します。

手順

Pod の Disruption Budget を設定するには、以下を実行します。

1. YAML ファイルを以下のようなオブジェクト定義で作成します。

```
apiVersion: policy/v1beta1 (1)
kind: PodDisruptionBudget
metadata:
  name: my-pdb
spec:
  minAvailable: 2 (2)
  selector: (3)
    matchLabels:
      foo: bar
```

(1) **PodDisruptionBudget** は **policy/v1beta1** API グループの一部です。

(2) 同時に利用可能である必要のある Pod の最小数。これには、整数またはパーセンテージ (例: **20%**) を指定する文字列を使用できます。

(3) 一連のリソースに対するラベルのクエリー。**matchLabels** と **matchExpressions** の結果は論理的に結合されます。

または、以下を実行します。

```
apiVersion: policy/v1beta1 (1)
kind: PodDisruptionBudget
metadata:
  name: my-pdb
spec:
  maxUnavailable: 25% (2)
  selector: (3)
    matchLabels:
      foo: bar
```

1. **PodDisruptionBudget** は **policy/v1beta1** API グループの一部です。
2. 同時に利用不可にできる Pod の最大数。これには、整数またはパーセンテージ (例: **20%**) を指定する文字列を使用できます。
3. 一連のリソースに対するラベルのクエリー。**matchLabels** と **matchExpressions** の結果は論理的に結合されます。

2. 以下のコマンドを実行してオブジェクトをプロジェクトに追加します。

```
$ oc create -f </path/to/file> -n <project_name>
```

2.3.4. Critical Pod の使用による Pod の削除の防止

クラスターを十分に機能させるために不可欠であるのに、マスターノードではなく通常のクラスターノードで実行される重要なコンポーネントは多数あります。重要なアドオンをエビクトすると、クラスターが正常に動作しなくなる可能性があります。

Critical とマークされている Pod はエビクトできません。

手順

Pod を Citical にするには、以下を実行します。

1. **Pod** 仕様を作成するか、または既存の Pod を編集して **system-cluster-critical** 優先順位クラスを含めます。

```
spec:
  template:
    metadata:
      name: critical-pod
    priorityClassName: system-cluster-critical 1
```

1. ノードからエビクトすべきではない Pod のデフォルトの優先順位クラス。

   または、クラスターにとって重要だが、必要に応じて削除できる Pod に **system-node-critical** を指定することもできます。

2. Pod を作成します。

```
$ oc create -f <file-name>.yaml
```

2.4. HORIZONTAL POD AUTOSCALER での POD の自動スケーリング

開発者として、Horizontal Pod Autoscaler (HPA) を使って、レプリケーションコントローラーに属する Pod から収集されるメトリクスまたはデプロイメント設定に基づき、OpenShift Container Platform がレプリケーションコントローラーまたはデプロイメント設定のスケールを自動的に増減する方法を指定できます。

2.4.1. Horizontal Pod Autoscaler について

Horizontal Pod Autoscaler を作成することで、実行する Pod の最小数と最大数を指定するだけでなく、Pod がターゲットに設定する CPU の使用率またはメモリー使用率を指定することができます。

Horizontal Pod Autoscaler を作成すると、OpenShift Container Platform は Pod で CPU またはメモリーリソースのメトリクスのクエリーを開始します。メトリクスが利用可能になると、Horizontal Pod Autoscaler は必要なメトリクスの使用率に対する現在のメトリクスの使用率の割合を計算し、随時スケールアップまたはスケールダウンを実行します。クエリーとスケーリングは一定間隔で実行されますが、メトリクスが利用可能になるでに 1 分から 2 分の時間がかかる場合があります。

レプリケーションコントローラーの場合、このスケーリングはレプリケーションコントローラーのレプリカに直接対応します。デプロイメント設定の場合、スケーリングはデプロイメント設定のレプリカ数に直接対応します。自動スケーリングは **Complete** フェーズの最新デプロイメントにのみ適用されることに注意してください。

OpenShift Container Platform はリソースに自動的に対応し、起動時などのリソースの使用が急増した場合など必要のない自動スケーリングを防ぎます。**unready** 状態の Pod には、スケールアップ時の使用率が **0 CPU** と指定され、Autoscaler はスケールダウン時にはこれらの Pod を無視します。既知のメトリクスのない Pod にはスケールアップ時の使用率が **0% CPU**、スケールダウン時に **100% CPU** となります。これにより、HPA の決定時に安定性が増します。この機能を使用するには、readiness チェックを設定して新規 Pod が使用可能であるかどうかを判別します。

Horizontal Pod Autoscaler を使用するには、クラスターの管理者はクラスターメトリクスを適切に設定している必要があります。

2.4.1.1. サポートされるメトリクス

以下のメトリクスは Horizontal Pod Autoscaler でサポートされています。

表2.1 メトリクス

メトリクス	説明	API バージョン
CPU の使用率	使用されている CPU コアの数。Pod の要求される CPU の割合の計算に使用されます。	**autoscaling/v1**、**autoscaling/v2beta2**
メモリーの使用率	使用されているメモリーの量。Pod の要求されるメモリーの割合の計算に使用されます。	**autoscaling/v2beta2**

重要

メモリーベースの自動スケーリングでは、メモリー使用量がレプリカ数と比例して増減する必要があります。平均的には以下のようになります。

- レプリカ数が増えると、Pod ごとのメモリー (作業セット) の使用量が全体的に減少します。
- レプリカ数が減ると、Pod ごとのメモリー使用量が全体的に増加します。

OpenShift Container Platform Web コンソールを使用して、アプリケーションのメモリー動作を確認し、メモリーベースの自動スケーリングを使用する前にアプリケーションがそれらの要件を満たしていることを確認します。

以下の例は、**image-registry DeploymentConfig** オブジェクトの自動スケーリングを示しています。最初のデプロイメントでは 3 つの Pod が必要です。HPA オブジェクトは最小で 5 まで増加され、Pod 上の CPU 使用率が 75% に達すると、Pod を最大 7 まで増やします。

```
$ oc autoscale dc/image-registry --min=5 --max=7 --cpu-percent=75
```

出力例

```
horizontalpodautoscaler.autoscaling/image-registry autoscaled
```

minReplicas が 3 に設定された image-registryDeploymentConfig オブジェクトのサンプル HPA

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: image-registry
  namespace: default
spec:
  maxReplicas: 7
  minReplicas: 3
  scaleTargetRef:
    apiVersion: apps.openshift.io/v1
    kind: DeploymentConfig
    name: image-registry
  targetCPUUtilizationPercentage: 75
status:
  currentReplicas: 5
  desiredReplicas: 0
```

1. デプロイメントの新しい状態を表示します。

   ```
   $ oc get dc image-registry
   ```

   デプロイメントには 5 つの Pod があります。

   出力例

   ```
   NAME             REVISION   DESIRED   CURRENT   TRIGGERED BY
   image-registry   1          5         5         config
   ```

2.4.1.2. スケーリングポリシー

autoscaling/v2beta2 API を使用すると、**スケーリングポリシー** を Horizontal Pod Autoscaler に追加できます。スケーリングポリシーは、OpenShift Container Platform の Horizontal Pod Autoscaler (HPA) が Pod をスケーリングする方法を制御します。スケーリングポリシーにより、特定の期間にスケーリングするように特定の数または特定のパーセンテージを設定して、HPA が Pod をスケールアップまたはスケールダウンするレートを制限できます。**固定化ウィンドウ (stabilization window)** を定義することもできます。これはメトリクスが変動する場合に、先に計算される必要な状態を使用してスケーリングを制御します。同じスケーリングの方向に複数のポリシーを作成し、変更の量に応じて使用するポリシーを判別することができます。タイミングが調整された反復によりスケーリングを制限することもできます。HPA は反復時に Pod をスケーリングし、その後の反復で必要に応じてスケーリングを実行します。

スケーリングポリシーを適用するサンプル HPA オブジェクト

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
```

```
metadata:
  name: hpa-resource-metrics-memory
  namespace: default
spec:
  behavior:
    scaleDown: 1
      policies: 2
      - type: Pods 3
        value: 4 4
        periodSeconds: 60 5
      - type: Percent
        value: 10 6
        periodSeconds: 60
      selectPolicy: Min 7
      stabilizationWindowSeconds: 300 8
    scaleUp: 9
      policies:
      - type: Pods
        value: 5 10
        periodSeconds: 70
      - type: Percent
        value: 12 11
        periodSeconds: 80
      selectPolicy: Max
      stabilizationWindowSeconds: 0
...
```

1. **scaleDown** または **scaleUp** のいずれかのスケーリングポリシーの方向を指定します。この例では、スケールダウンのポリシーを作成します。
2. スケーリングポリシーを定義します。
3. ポリシーが反復時に特定の Pod の数または Pod のパーセンテージに基づいてスケーリングするかどうかを決定します。デフォルト値は **pods** です。
4. 反復ごとに Pod の数または Pod のパーセンテージのいずれかでスケーリングの量を決定します。Pod 数でスケールダウンする際のデフォルト値はありません。
5. スケーリングの反復の長さを決定します。デフォルト値は **15** 秒です。
6. パーセンテージでのスケールダウンのデフォルト値は 100% です。
7. 複数のポリシーが定義されている場合は、最初に使用するポリシーを決定します。最大限の変更を許可するポリシーを使用するように **Max** を指定するか、最小限の変更を許可するポリシーを使用するように **Min** を指定するか、または HPA がポリシーの方向でスケーリングしないように **Disabled** を指定します。デフォルト値は **Max** です。
8. HPA が必要とされる状態で遡る期間を決定します。デフォルト値は **0** です。
9. この例では、スケールアップのポリシーを作成します。
10. Pod 数によるスケールアップの量。Pod 数をスケールアップするためのデフォルト値は 4% です。
11. Pod のパーセンテージによるスケールアップの量。パーセンテージでスケールアップするためのデフォルト値は 100% です。

スケールダウンポリシーの例

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-resource-metrics-memory
  namespace: default
spec:
...
  minReplicas: 20
...
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 300
      policies:
      - type: Pods
        value: 4
        periodSeconds: 30
      - type: Percent
        value: 10
        periodSeconds: 60
      selectPolicy: Max
    scaleUp:
      selectPolicy: Disabled
```

この例では、Pod の数が 40 より大きい場合、パーセントベースのポリシーがスケールダウンに使用されます。このポリシーでは、 **selectPolicy** による要求により、より大きな変更が生じるためです。

80 の Pod レプリカがある場合、初回の反復で HPA は Pod を 8 Pod 減らします。これは、1 分間 (**periodSeconds: 60**) の (**type: Percent** および **value: 10** パラメーターに基づく) 80 Pod の 10% に相当します。次回の反復では、Pod 数は 72 になります。HPA は、残りの Pod の 10% が 7.2 であると計算し、これを 8 に丸め、8 Pod をスケールダウンします。後続の反復ごとに、スケーリングされる Pod 数は残りの Pod 数に基づいて再計算されます。Pod の数が 40 未満の場合、Pod ベースの数がパーセントベースの数よりも大きくなるため、Pod ベースのポリシーが適用されます。HPA は、残りのレプリカ (**minReplicas**) が 20 になるまで、30 秒 (**periodSeconds: 30**) で一度に 4 Pod (**type: Pods** および **value: 4**) を減らします。

selectPolicy: Disabled パラメーターは HPA による Pod のスケールアップを防ぎます。必要な場合は、レプリカセットまたはデプロイメントセットでレプリカの数を調整して手動でスケールアップできます。

設定されている場合、**oc edit** コマンドを使用してスケーリングポリシーを表示できます。

```
$ oc edit hpa hpa-resource-metrics-memory
```

出力例

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  annotations:
    autoscaling.alpha.kubernetes.io/behavior:\
'{"ScaleUp":{"StabilizationWindowSeconds":0,"SelectPolicy":"Max","Policies":
[{"Type":"Pods","Value":4,"PeriodSeconds":15},{"Type":"Percent","Value":100,"PeriodSeconds":15}]},\
```

```
"ScaleDown":{"StabilizationWindowSeconds":300,"SelectPolicy":"Min","Policies":
[{"Type":"Pods","Value":4,"PeriodSeconds":60},{"Type":"Percent","Value":10,"PeriodSeconds":60}]}}'
...
```

2.4.2. Web コンソールを使用した Horizontal Pod Autoscaler の作成

Web コンソールから、デプロイメントで実行する Pod の最小および最大数を指定する Horizontal Pod Autoscaler (HPA) を作成できます。Pod がターゲットに設定する CPU またはメモリー使用量を定義することもできます。

注記

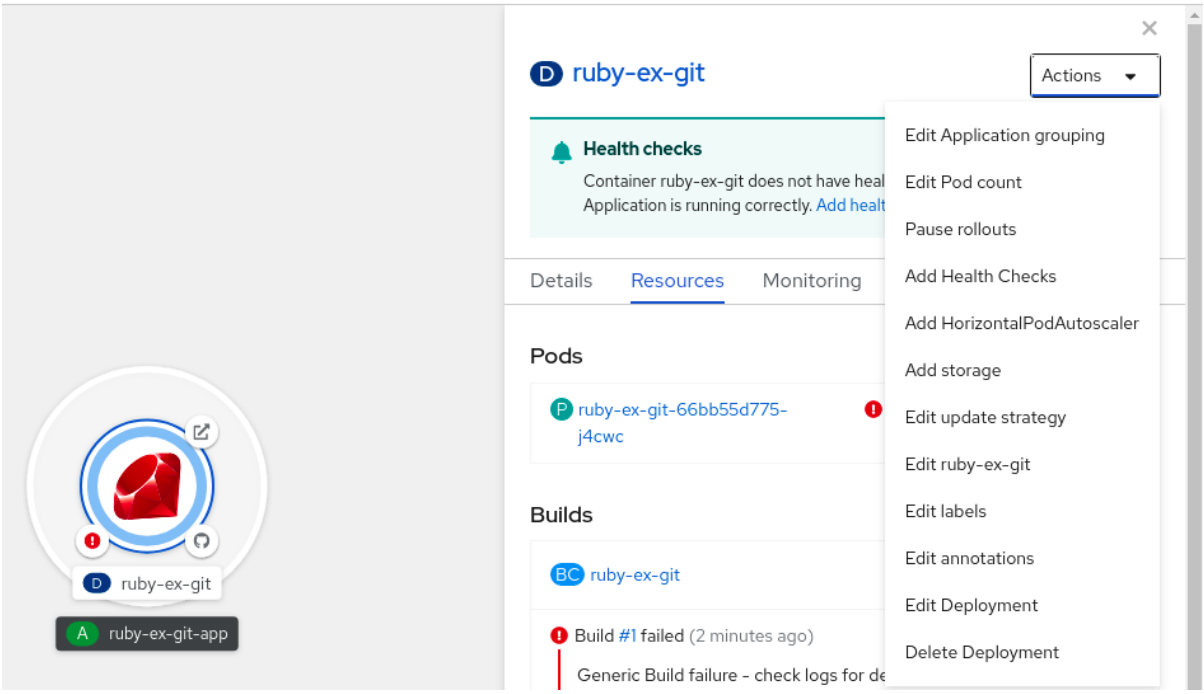
HPA は、Operator がサポートするサービス、Knative サービス、または Helm チャートの一部であるデプロイメントに追加することはできません。

手順

Web コンソールで HPA を作成するには、以下を実行します。

1. **Topology** ビューで、ノードをクリックしてサイドペインを表示します。
2. **Actions** ドロップダウンリストから、**Add HorizontalPodAutoscaler** を選択して **Add HorizontalPodAutoscaler** フォームを開きます。

図2.1 Horizontal Pod Autoscaler の追加



3. **Add HorizontalPodAutoscaler** フォームから、名前、最小および最大の Pod 制限、CPU およびメモリーの使用状況を定義し、**Save** をクリックします。

注記

CPU およびメモリー使用量の値のいずれかが見つからない場合は、警告が表示されます。

Web コンソールで HPA を編集するには、以下を実行します。

1. **Topology** ビューで、ノードをクリックしてサイドペインを表示します。
2. **Actions** ドロップダウンリストから、**Edit HorizontalPodAutoscaler** を選択し、 **Horizontal Pod Autoscaler** フォームを開きます。
3. **Edit Horizontal Pod Autoscaler** フォームから、最小および最大の Pod 制限および CPU およびメモリー使用量を編集し、**Save** をクリックします。

注記

Web コンソールで Horizontal Pod Autoscaler を作成または編集する際に、**Form view** から **YAML view**に切り替えることができます。

Web コンソールで HPA を削除するには、以下を実行します。

1. **Topology** ビューで、ノードをクリックし、サイドパネルを表示します。
2. **Actions** ドロップダウンリストから、**Remove HorizontalPodAutoscaler** を選択します。
3. 確認のポップアップウィンドウで、**Remove** をクリックして HPA を削除します。

2.4.3. CLI を使用した CPU 使用率向けの Horizontal Pod Autoscaler の作成

既存の **Deployment**、**DeploymentConfig**、**ReplicaSet**、**ReplicationController** 、または **StatefulSet** オブジェクトの水平 Pod オートスケーラー (HPA) を作成して、そのオブジェクトに関連付けられた Pod を自動的にスケーリングし、指定した CPU 使用率を維持できます。

HPA は、すべての Pod で指定された CPU 使用率を維持するために、最小数と最大数の間でレプリカ数を増減します。

CPU 使用率について自動スケーリングを行う際に、**oc autoscale** コマンドを使用し、実行する必要のある Pod の最小数および最大数と Pod がターゲットとして設定する必要のある平均 CPU 使用率を指定することができます。最小値を指定しない場合、Pod には OpenShift Container Platform サーバーからのデフォルト値が付与されます。特定の CPU 値について自動スケーリングを行うには、ターゲット CPU および Pod の制限のある **HorizontalPodAutoscaler** オブジェクトを作成します。

前提条件

Horizontal Pod Autoscaler を使用するには、クラスターの管理者はクラスターメトリクスを適切に設定している必要があります。メトリクスが設定されているかどうかは、**oc describe PodMetrics <pod-name>** コマンドを使用して判断できます。メトリクスが設定されている場合、出力は以下の **Usage** の下にある **Cpu** と **Memory** のように表示されます。

```
$ oc describe PodMetrics openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
```

出力例

```
Name:         openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
Namespace:    openshift-kube-scheduler
Labels:       <none>
Annotations:  <none>
API Version:  metrics.k8s.io/v1beta1
Containers:
  Name:  wait-for-host-port
  Usage:
```

```
    Memory:  0
  Name:      scheduler
  Usage:
    Cpu:     8m
    Memory:  45440Ki
Kind:        PodMetrics
Metadata:
  Creation Timestamp:  2019-05-23T18:47:56Z
  Self Link:           /apis/metrics.k8s.io/v1beta1/namespaces/openshift-kube-scheduler/pods/openshift-
kube-scheduler-ip-10-0-135-131.ec2.internal
Timestamp:             2019-05-23T18:47:56Z
Window:                1m0s
Events:                <none>
```

手順

CPU 使用率のための Horizontal Pod Autoscaler を作成するには、以下を実行します。

1. 以下のいずれかを実行します。

   - CPU 使用率のパーセントに基づいてスケーリングするには、既存のオブジェクトとして **HorizontalPodAutoscaler** オブジェクトを作成します。

     ```
     $ oc autoscale <object_type>/<name> \ (1)
       --min <number> \ (2)
       --max <number> \ (3)
       --cpu-percent=<percent> (4)
     ```

     (1) 自動スケーリングするオブジェクトのタイプと名前を指定します。オブジェクトが存在し、**Deployment**、**DeploymentConfig/dc**、**ReplicaSet/rs**、**ReplicationController/rc**、または **StatefulSet** である必要があります。

     (2) オプションで、スケールダウン時のレプリカの最小数を指定します。

     (3) スケールアップ時のレプリカの最大数を指定します。

     (4) 要求された CPU のパーセントで表示された、すべての Pod に対する目標の平均 CPU 使用率を指定します。指定しない場合または負の値の場合、デフォルトの自動スケーリングポリシーが使用されます。

     たとえば、次のコマンドが示すように、**image-registryDeploymentConfig** オブジェクトの自動スケーリング。最初のデプロイメントでは 3 つの Pod が必要です。HPA オブジェクトは最小で 5 まで増加され、Pod 上の CPU 使用率が 75% に達すると、Pod を最大 7 まで増やします。

     ```
     $ oc autoscale dc/image-registry --min=5 --max=7 --cpu-percent=75
     ```

   - 特定の CPU 値に合わせてスケーリングするには、既存のオブジェクトに対して次のような YAML ファイルを作成します。

     a. 以下のような YAML ファイルを作成します。

     ```
     apiVersion: autoscaling/v2beta2 (1)
     kind: HorizontalPodAutoscaler
     ```

```
metadata:
  name: cpu-autoscale (2)
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1 (3)
    kind: ReplicaSet (4)
    name: example (5)
  minReplicas: 1 (6)
  maxReplicas: 10 (7)
  metrics: (8)
  - type: Resource
    resource:
      name: cpu (9)
      target:
        type: AverageValue (10)
        averageValue: 500m (11)
```

1. **autoscaling/v2beta2** API を使用します。
2. この Horizontal Pod Autoscaler オブジェクトの名前を指定します。
3. スケーリングするオブジェクトの API バージョンを指定します。
   - **ReplicationController** の場合は、 **v1** を使用します。
   - **DeploymentConfig** の場合は、 **apps.openshift.io/v1** を使用します。
   - **Deployment**、**ReplicaSet**、**Statefulset** オブジェクトの場合は、**apps/v1** を使用します。
4. オブジェクトのタイプを指定します。オブジェクトは、**Deployment**、**DeploymentConfig/dc**、**ReplicaSet/rs**、**ReplicationController/rc** 、または **StatefulSet** である必要があります。
5. スケーリングするオブジェクトの名前を指定します。オブジェクトが存在する必要があります。
6. スケールダウン時のレプリカの最小数を指定します。
7. スケールアップ時のレプリカの最大数を指定します。
8. メモリー使用率に **metrics** パラメーターを使用します。
9. CPU 使用率に **cpu** を指定します。
10. **AverageValue** に設定します。
11. ターゲットに設定された CPU 値で **averageValue** に設定します。

b. Horizontal Pod Autoscaler を作成します。

```
$ oc create -f <file-name>.yaml
```

2. Horizontal Pod Autoscaler が作成されていることを確認します。

```
$ oc get hpa cpu-autoscale
```

出力例

```
NAME          REFERENCE                    TARGETS       MINPODS   MAXPODS
REPLICAS   AGE
cpu-autoscale   ReplicationController/example   173m/500m      1         10        1          20m
```

2.4.4. CLI を使用したメモリー使用率向けの Horizontal Pod Autoscaler オブジェクトの作成

直接の値または要求されるメモリーのパーセンテージのいずれかで指定する平均のメモリー使用率を維持するために、オブジェクトに関連付けられた Pod を自動的にスケーリングする既存の **DeploymentConfig** オブジェクトまたは **ReplicationController** オブジェクトの Horizontal Pod Autoscaler (HPA) を作成できます。

HPA は、すべての Pod で指定のメモリー使用率を維持するために、最小数と最大数の間のレプリカ数を増減します。

メモリー使用率については、Pod の最小数および最大数と、Pod がターゲットとする平均のメモリー使用率を指定することができます。最小値を指定しない場合、Pod には OpenShift Container Platform サーバーからのデフォルト値が付与されます。

前提条件

Horizontal Pod Autoscaler を使用するには、クラスターの管理者はクラスターメトリクスを適切に設定している必要があります。メトリクスが設定されているかどうかは、**oc describe PodMetrics <pod-name>** コマンドを使用して判断できます。メトリクスが設定されている場合、出力は以下の **Usage** の下にある **Cpu** と **Memory** のように表示されます。

```
$ oc describe PodMetrics openshift-kube-scheduler-ip-10-0-129-223.compute.internal -n openshift-kube-scheduler
```

出力例

```
Name:         openshift-kube-scheduler-ip-10-0-129-223.compute.internal
Namespace:    openshift-kube-scheduler
Labels:       <none>
Annotations:  <none>
API Version:  metrics.k8s.io/v1beta1
Containers:
  Name:  scheduler
  Usage:
    Cpu:     2m
    Memory:  41056Ki
  Name:      wait-for-host-port
  Usage:
    Memory:  0
Kind:        PodMetrics
Metadata:
  Creation Timestamp:  2020-02-14T22:21:14Z
  Self Link:           /apis/metrics.k8s.io/v1beta1/namespaces/openshift-kube-scheduler/pods/openshift-
```

```
kube-scheduler-ip-10-0-129-223.compute.internal
Timestamp:           2020-02-14T22:21:14Z
Window:              5m0s
Events:              <none>
```

手順

メモリー使用率の Horizontal Pod Autoscaler を作成するには、以下を実行します。

1. 以下のいずれか 1 つを含む YAML ファイルを作成します。

   - 特定のメモリー値に合わせてスケーリングするには、既存の **ReplicationController** オブジェクトまたはレプリケーションコントローラーに対して、次のような **HorizontalPodAutoscaler** オブジェクトを作成します。

     出力例

```
apiVersion: autoscaling/v2beta2 (1)
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-resource-metrics-memory (2)
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: v1 (3)
    kind: ReplicationController (4)
    name: example (5)
  minReplicas: 1 (6)
  maxReplicas: 10 (7)
  metrics: (8)
  - type: Resource
    resource:
      name: memory (9)
      target:
        type: AverageValue (10)
        averageValue: 500Mi (11)
  behavior: (12)
    scaleDown:
      stabilizationWindowSeconds: 300
      policies:
      - type: Pods
        value: 4
        periodSeconds: 60
      - type: Percent
        value: 10
        periodSeconds: 60
      selectPolicy: Max
```

(1) **autoscaling/v2beta2** API を使用します。

(2) この Horizontal Pod Autoscaler オブジェクトの名前を指定します。

(3) スケーリングするオブジェクトの API バージョンを指定します。

- レプリケーションコントローラーについては、**v1** を使用します。
- **DeploymentConfig** オブジェクトについては、**apps.openshift.io/v1** を使用します。

4 スケーリングするオブジェクトの種類 (**ReplicationController** または **DeploymentConfig** のいずれか) を指定します。

5 スケーリングするオブジェクトの名前を指定します。オブジェクトが存在する必要があります。

6 スケールダウン時のレプリカの最小数を指定します。

7 スケールアップ時のレプリカの最大数を指定します。

8 メモリー使用率に **metrics** パラメーターを使用します。

9 メモリー使用率の **memory** を指定します。

10 タイプを **AverageValue** に設定します。

11 **averageValue** および特定のメモリー値を指定します。

12 オプション: スケールアップまたはスケールダウンのレートを制御するスケーリングポリシーを指定します。

- パーセンテージについてスケーリングするには、以下のように **HorizontalPodAutoscaler** オブジェクトを作成します。

出力例

```
apiVersion: autoscaling/v2beta2 1
kind: HorizontalPodAutoscaler
metadata:
  name: memory-autoscale 2
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps.openshift.io/v1 3
    kind: DeploymentConfig 4
    name: example 5
  minReplicas: 1 6
  maxReplicas: 10 7
  metrics: 8
  - type: Resource
    resource:
      name: memory 9
      target:
        type: Utilization 10
        averageUtilization: 50 11
  behavior: 12
    scaleUp:
      stabilizationWindowSeconds: 180
      policies:
```

```
    - type: Pods
      value: 6
      periodSeconds: 120
    - type: Percent
      value: 10
      periodSeconds: 120
    selectPolicy: Max
```

1. **autoscaling/v2beta2** API を使用します。
2. この Horizontal Pod Autoscaler オブジェクトの名前を指定します。
3. スケーリングするオブジェクトの API バージョンを指定します。
   - レプリケーションコントローラーについては、**v1** を使用します。
   - **DeploymentConfig** オブジェクトについては、**apps.openshift.io/v1** を使用します。
4. スケーリングするオブジェクトの種類 (**ReplicationController** または **DeploymentConfig** のいずれか) を指定します。
5. スケーリングするオブジェクトの名前を指定します。オブジェクトが存在する必要があります。
6. スケールダウン時のレプリカの最小数を指定します。
7. スケールアップ時のレプリカの最大数を指定します。
8. メモリー使用率に **metrics** パラメーターを使用します。
9. メモリー使用率の **memory** を指定します。
10. **Utilization** に設定します。
11. **averageUtilization** および ターゲットに設定する平均メモリー使用率をすべての Pod に対して指定します (要求されるメモリーのパーセントで表す)。ターゲット Pod にはメモリー要求が設定されている必要があります。
12. オプション: スケールアップまたはスケールダウンのレートを制御するスケーリングポリシーを指定します。

2. Horizontal Pod Autoscaler を作成します。

```
$ oc create -f <file-name>.yaml
```

以下に例を示します。

```
$ oc create -f hpa.yaml
```

出力例

```
horizontalpodautoscaler.autoscaling/hpa-resource-metrics-memory created
```

3. Horizontal Pod Autoscaler が作成されていることを確認します。

```
$ oc get hpa hpa-resource-metrics-memory
```

出力例

```
NAME                          REFERENCE                          TARGETS        MINPODS   MAXPODS
REPLICAS   AGE
hpa-resource-metrics-memory   ReplicationController/example   2441216/500Mi   1         10
1          20m
```

```
$ oc describe hpa hpa-resource-metrics-memory
```

出力例

```
Name:                       hpa-resource-metrics-memory
Namespace:                  default
Labels:                     <none>
Annotations:                <none>
CreationTimestamp:          Wed, 04 Mar 2020 16:31:37 +0530
Reference:                  ReplicationController/example
Metrics:                    ( current / target )
  resource memory on pods:   2441216 / 500Mi
Min replicas:               1
Max replicas:               10
ReplicationController pods:  1 current / 1 desired
Conditions:
  Type            Status  Reason              Message
  ----            ------  ------              -------
  AbleToScale     True    ReadyForNewScale    recommended size matches current size
  ScalingActive   True    ValidMetricFound    the HPA was able to successfully calculate a
replica count from memory resource
  ScalingLimited  False   DesiredWithinRange  the desired count is within the acceptable
range
Events:
  Type     Reason                 Age               From                       Message
  ----     ------                 ----              ----                       -------
  Normal   SuccessfulRescale      6m34s             horizontal-pod-autoscaler  New size: 1;
reason: All metrics below target
```

2.4.5. CLI を使用した Horizontal Pod Autoscaler の状態条件について

状態条件セットを使用して、Horizontal Pod Autoscaler (HPA) がスケーリングできるかどうかや、現時点でこれがいずれかの方法で制限されているかどうかを判別できます。

HPA の状態条件は、自動スケーリング API の **v2beta1** バージョンで利用できます。

HPA は、以下の状態条件で応答します。

- **AbleToScale** 条件では、HPA がメトリクスを取得して更新できるか、またバックオフ関連の条件によりスケーリングが回避されるかどうかを指定します。
  - **True** 条件はスケーリングが許可されることを示します。

- **False** 条件は指定される理由によりスケーリングが許可されないことを示します。

- **ScalingActive** 条件は、HPA が有効にされており (ターゲットのレプリカ数がゼロでない)、必要なメトリクスを計算できるかどうかを示します。
  - **True** 条件はメトリクスが適切に機能していることを示します。
  - **False** 条件は通常フェッチするメトリクスに関する問題を示します。

- **ScalingLimited** 条件は、必要とするスケールが Horizontal Pod Autoscaler の最大値または最小値によって制限されていたことを示します。
  - **True** 条件は、スケーリングするためにレプリカの最小または最大数を引き上げるか、または引き下げる必要があることを示します。
  - **False** 条件は、要求されたスケーリングが許可されることを示します。

```
$ oc describe hpa cm-test
```

出力例

```
Name:                       cm-test
Namespace:                  prom
Labels:                     <none>
Annotations:                <none>
CreationTimestamp:          Fri, 16 Jun 2017 18:09:22 +0000
Reference:                  ReplicationController/cm-test
Metrics:                    ( current / target )
  "http_requests" on pods:      66m / 500m
Min replicas:               1
Max replicas:               4
ReplicationController pods:     1 current / 1 desired
Conditions: 1
  Type            Status    Reason              Message
  ----            ------    ------              -------
  AbleToScale       True      ReadyForNewScale    the last scale time was sufficiently old as to warrant a new scale
  ScalingActive     True      ValidMetricFound    the HPA was able to successfully calculate a replica count from pods metric http_request
  ScalingLimited    False     DesiredWithinRange  the desired replica count is within the acceptable range
Events:
```

1 Horizontal Pod Autoscaler の状況メッセージです。

以下は、スケーリングできない Pod の例です。

出力例

```
Conditions:
  Type        Status  Reason          Message
  ----        ------  ------          -------
  AbleToScale  False   FailedGetScale  the HPA controller was unable to get the target's current scale: no matches for kind "ReplicationController" in group "apps"
```

```
Events:
  Type     Reason          Age              From                    Message
  ----     ------          ----             ----                    -------
  Warning  FailedGetScale  6s (x3 over 36s)  horizontal-pod-autoscaler  no matches for kind
"ReplicationController" in group "apps"
```

以下は、スケーリングに必要なメトリクスを取得できなかった Pod の例です。

出力例

```
Conditions:
  Type               Status    Reason                  Message
  ----               ------    ------                  -------
  AbleToScale        True      SucceededGetScale       the HPA controller was able to get the target's
current scale
  ScalingActive      False     FailedGetResourceMetric    the HPA was unable to compute the replica
count: failed to get cpu utilization: unable to get metrics for resource cpu: no metrics returned from
resource metrics API
```

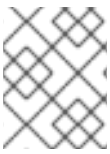
以下は、要求される自動スケーリングが要求される最小数よりも小さい場合の Pod の例です。

出力例

```
Conditions:
  Type            Status    Reason            Message
  ----            ------    ------            -------
  AbleToScale     True      ReadyForNewScale    the last scale time was sufficiently old as to warrant
a new scale
  ScalingActive   True      ValidMetricFound    the HPA was able to successfully calculate a replica
count from pods metric http_request
  ScalingLimited  False     DesiredWithinRange  the desired replica count is within the acceptable
range
```

2.4.5.1. CLI を使用した Horizontal Pod Autoscaler の状態条件の表示

Pod に設定された状態条件は、Horizontal Pod Autoscaler (HPA) で表示することができます。



注記

Horizontal Pod Autoscaler の状態条件は、自動スケーリング API の **v2beta1** バージョンで利用できます。

前提条件

Horizontal Pod Autoscaler を使用するには、クラスターの管理者はクラスターメトリクスを適切に設定している必要があります。メトリクスが設定されているかどうかは、**oc describe PodMetrics <pod-name>** コマンドを使用して判断できます。メトリクスが設定されている場合、出力は以下の **Usage** の下にある **Cpu** と **Memory** のように表示されます。

```
$ oc describe PodMetrics openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
```

出力例

```
Name:         openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
Namespace:    openshift-kube-scheduler
Labels:       <none>
Annotations:  <none>
API Version:  metrics.k8s.io/v1beta1
Containers:
  Name:  wait-for-host-port
  Usage:
    Memory:  0
  Name:      scheduler
  Usage:
    Cpu:     8m
    Memory:  45440Ki
Kind:        PodMetrics
Metadata:
  Creation Timestamp:  2019-05-23T18:47:56Z
  Self Link:           /apis/metrics.k8s.io/v1beta1/namespaces/openshift-kube-scheduler/pods/openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
Timestamp:             2019-05-23T18:47:56Z
Window:                1m0s
Events:                <none>
```

手順

Pod の状態条件を表示するには、Pod の名前と共に以下のコマンドを使用します。

```
$ oc describe hpa <pod-name>
```

以下に例を示します。

```
$ oc describe hpa cm-test
```

条件は、出力の **Conditions** フィールドに表示されます。

出力例

```
Name:                           cm-test
Namespace:                         prom
Labels:                         <none>
Annotations:                    <none>
CreationTimestamp:              Fri, 16 Jun 2017 18:09:22 +0000
Reference:                      ReplicationController/cm-test
Metrics:                        ( current / target )
  "http_requests" on pods:      66m / 500m
Min replicas:                   1
Max replicas:                   4
ReplicationController pods:     1 current / 1 desired
Conditions: (1)
  Type            Status    Reason              Message
  ----            ------    ------              -------
  AbleToScale     True      ReadyForNewScale    the last scale time was sufficiently old as to warrant a new scale
  ScalingActive   True      ValidMetricFound    the HPA was able to successfully calculate a replica
```

```
count from pods metric http_request
  ScalingLimited    False    DesiredWithinRange  the desired replica count is within the acceptable
range
```

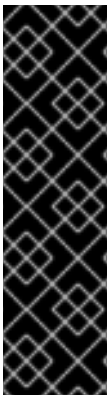
2.4.6. 関連情報

- レプリケーションコントローラーとデプロイメントコントローラーの詳細は、デプロイメントおよびデプロイメント設定 について参照してください。

2.5. VERTICAL POD AUTOSCALER を使用した POD リソースレベルの自動調整

OpenShift Container Platform の Vertical Pod Autoscaler Operator (VPA) は、Pod 内のコンテナーの履歴および現在の CPU とメモリーリソースを自動的に確認し、把握する使用値に基づいてリソース制限および要求を更新できます。VPA は個別のカスタムリソース (CR) を使用して、プロジェクトの **Deployment**、**Deployment Config**、**StatefulSet**、**Job**、**DaemonSet**、**ReplicaSet**、または **ReplicationController** などのワークロードオブジェクトに関連付けられたすべての Pod を更新します。

VPA は、Pod に最適な CPU およびメモリーの使用状況を理解するのに役立ち、Pod のライフサイクルを通じて Pod のリソースを自動的に維持します。



重要

Vertical Pod Autoscaler はテクノロジープレビュー機能としてのみご利用いただけます。テクノロジープレビュー機能は Red Hat の実稼働環境でのサービスレベルアグリーメント (SLA) ではサポートされていないため、Red Hat では実稼働環境での使用を推奨していません。Red Hat は実稼働環境でこれらを使用することを推奨していません。テクノロジープレビューの機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行いフィードバックを提供していただくことを目的としています。

Red Hat のテクノロジープレビュー機能のサポート範囲についての詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

2.5.1. Vertical Pod Autoscaler Operator について

Vertical Pod Autoscaler Operator (VPA) は、API リソースおよびカスタムリソース (CR) として実装されます。CR は、プロジェクトのデーモンセット、レプリケーションコントローラーなどの特定のワークロードオブジェクトに関連付けられた Pod について Vertical Pod Autoscaler Operator が取るべき動作を判別します。

VPA は、それらの Pod 内のコンテナーの履歴および現在の CPU とメモリーの使用状況を自動的に計算し、このデータを使用して、最適化されたリソース制限および要求を判別し、これらの Pod が常時効率的に動作していることを確認することができます。たとえば、VPA は使用している量よりも多くのリソースを要求する Pod のリソースを減らし、十分なリソースを要求していない Pod のリソースを増やします。

VPA は、一度に 1 つずつ推奨値で調整されていない Pod を自動的に削除するため、アプリケーションはダウンタイムなしに継続して要求を提供できます。ワークロードオブジェクトは、元のリソース制限および要求で Pod を再デプロイします。VPA は変更用の受付 Webhook を使用して、Pod がノードに許可される前に最適化されたリソース制限および要求で Pod を更新します。VPA が Pod を削除する必要がない場合は、VPA リソース制限および要求を表示し、必要に応じて Pod を手動で更新できます。

たとえば、CPU の 50% を使用する Pod が 10% しか要求しない場合、VPA は Pod が要求よりも多くの CPU を消費すると判別してその Pod を削除します。レプリカセットなどのワークロードオブジェクトは Pod を再起動し、VPA は推奨リソースで新しい Pod を更新します。

開発者の場合、VPA を使用して、Pod を各 Pod に適したリソースを持つノードにスケジュールし、Pod の需要の多い期間でも稼働状態を維持することができます。

管理者は、VPA を使用してクラスターリソースをより適切に活用できます。たとえば、必要以上の CPU リソースを Pod が予約できないようにします。VPA は、ワークロードが実際に使用しているリソースをモニターし、他のワークロードで容量を使用できるようにリソース要件を調整します。VPA は、初期のコンテナー設定で指定される制限と要求の割合をそのまま維持します。

注記

VPA の実行を停止するか、またはクラスターの特定の VPA CR を削除する場合、VPA によってすでに変更された Pod のリソース要求は変更されません。新規 Pod は、VPA による以前の推奨事項ではなく、ワークロードオブジェクトで定義されたリソースを取得します。

2.5.2. Vertical Pod Autoscaler Operator のインストール

OpenShift Container Platform Web コンソールを使って Vertical Pod Autoscaler Operator (VPA) をインストールすることができます。

手順

1. OpenShift Container Platform Web コンソールで、**Operators → OperatorHub** をクリックします。
2. 利用可能な Operator の一覧から **VerticalPodAutoscaler** を選択し、**Install** をクリックします。
3. **Install Operator** ページで、**Operator recommended namespace** オプションが選択されていることを確認します。これにより、Operator が必須の **openshift-vertical-pod-autoscaler** namespace にインストールされます。この namespace は存在しない場合は、自動的に作成されます。
4. **Install** をクリックします。
5. VPA Operator コンポーネントを一覧表示して、インストールを確認します。
   a. **Workloads → Pods** に移動します。
   b. ドロップダウンメニューから **openshift-vertical-pod-autoscaler** プロジェクトを選択し、4 つの Pod が実行されていることを確認します。
   c. **Workloads → Deployments**に移動し、4 つの デプロイメントが実行されていることを確認します。
6. オプション:以下のコマンドを使用して、OpenShift Container Platform CLI でインストールを確認します。

```
$ oc get all -n openshift-vertical-pod-autoscaler
```

出力には、4 つの Pod と 4 つのデプロイメントが表示されます。

出力例

```
NAME                                                READY   STATUS    RESTARTS   AGE
pod/vertical-pod-autoscaler-operator-85b4569c47-2gmhc   1/1     Running   0          3m13s
pod/vpa-admission-plugin-default-67644fc87f-xq7k9       1/1     Running   0          2m56s
pod/vpa-recommender-default-7c54764b59-8gckt            1/1     Running   0          2m56s
pod/vpa-updater-default-7f6cc87858-47vw9                1/1     Running   0          2m56s

NAME                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
service/vpa-webhook   ClusterIP   172.30.53.206   <none>        443/TCP   2m56s

NAME                                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/vertical-pod-autoscaler-operator   1/1     1            1           3m13s
deployment.apps/vpa-admission-plugin-default       1/1     1            1           2m56s
deployment.apps/vpa-recommender-default            1/1     1            1           2m56s
deployment.apps/vpa-updater-default                1/1     1            1           2m56s

NAME                                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/vertical-pod-autoscaler-operator-85b4569c47   1         1         1       3m13s
replicaset.apps/vpa-admission-plugin-default-67644fc87f       1         1         1       2m56s
replicaset.apps/vpa-recommender-default-7c54764b59            1         1         1       2m56s
replicaset.apps/vpa-updater-default-7f6cc87858                1         1         1       2m56s
```

2.5.3. Vertical Pod Autoscaler Operator の使用について

Vertical Pod Autoscaler Operator (VPA) を使用するには、クラスター内にワークロードオブジェクトの VPA カスタムリソース (CR) を作成します。VPA は、そのワークロードオブジェクトに関連付けられた Pod に最適な CPU およびメモリーリソースを確認し、適用します。VPA は、デプロイメント、ステートフルセット、ジョブ、デーモンセット、レプリカセット、またはレプリケーションコントローラーのワークロードオブジェクトと共に使用できます。VPA CR はモニターする必要のある Pod と同じプロジェクトになければなりません。

VPA CR を使用してワークロードオブジェクトを関連付け、VPA が動作するモードを指定します。

- **Auto** および **Recreate** モードは、Pod の有効期間中は VPA CPU およびメモリーの推奨事項を自動的に適用します。VPA は、推奨値で調整されていないプロジェクトの Pod を削除します。ワークロードオブジェクトによって再デプロイされる場合、VPA はその推奨内容で新規 Pod を更新します。
- **Initial** モードは、Pod の作成時にのみ VPA の推奨事項を自動的に適用します。
- **Off** モードは、推奨されるリソース制限および要求のみを提供するので、推奨事項を手動で適用することができます。**off** モードは Pod を更新しません。

CR を使用して、VPA 評価および更新から特定のコンテナーをオプトアウトすることもできます。

たとえば、Pod には以下の制限および要求があります。

```
resources:
  limits:
    cpu: 1
    memory: 500Mi
  requests:
    cpu: 500m
    memory: 100Mi
```

auto に設定された VPA を作成すると、VPA はリソースの使用状況を確認して Pod を削除します。再デプロイ時に、Pod は新規のリソース制限および要求を使用します。

```
resources:
  limits:
    cpu: 50m
    memory: 1250Mi
  requests:
    cpu: 25m
    memory: 262144k
```

以下のコマンドを実行して、VPA の推奨事項を表示できます。

```
$ oc get vpa <vpa-name> --output yaml
```

数分後に、出力には、以下のような CPU およびメモリー要求の推奨内容が表示されます。

出力例

```
...
status:
...
  recommendation:
    containerRecommendations:
    - containerName: frontend
      lowerBound:
        cpu: 25m
        memory: 262144k
      target:
        cpu: 25m
        memory: 262144k
      uncappedTarget:
        cpu: 25m
        memory: 262144k
      upperBound:
        cpu: 262m
        memory: "274357142"
    - containerName: backend
      lowerBound:
        cpu: 12m
        memory: 131072k
      target:
        cpu: 12m
        memory: 131072k
      uncappedTarget:
        cpu: 12m
        memory: 131072k
      upperBound:
        cpu: 476m
        memory: "498558823"
...
```

出力には、**target** (推奨リソース)、**lowerBound** (最小推奨リソース)、**upperBound** (最大推奨リソース)、および **uncappedTarget** (最新の推奨リソース) が表示されます。

VPA は **lowerBound** および **upperBound** の値を使用して、Pod の更新が必要であるかどうかを判別します。Pod のリソース要求が **lowerBound** 値を下回るか、**upperBound** 値を上回る場合は、VPA は終了し、**target** 値で Pod を再作成します。

2.5.3.1. VPA の推奨事項の自動適用

VPA を使用して Pod を自動的に更新するには、**updateMode** が **Auto** または **Recreate** に設定された特定のワークロードオブジェクトの VPA CR を作成します。

Pod がワークロードオブジェクト用に作成されると、VPA はコンテナーを継続的にモニターして、CPU およびメモリーのニーズを分析します。VPA は、CPU およびメモリーについての VPA の推奨値を満たさない Pod を削除します。再デプロイ時に、Pod は VPA の推奨値に基づいて新規のリソース制限および要求を使用し、アプリケーションに設定された Pod の Disruption Budget (停止状態の予算) を反映します。この推奨事項は、参照用に VPA CR の **status** フィールドに追加されます。

注記

ワークロードオブジェクトは、VPA が Pod を監視し、更新できるようにレプリカを 2 つ以上指定する必要があります。ワークロードオブジェクトが 1 つのレプリカを指定する場合、VPA はアプリケーションのダウンタイムを防ぐために Pod を削除しません。Pod を手動で削除し、推奨リソースを使用することができます。

Auto モードの VPA CR の例

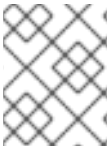
```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment 1
    name:       frontend 2
  updatePolicy:
    updateMode: "Auto" 3
```

1 1 この VPA CR が管理するワークロードオブジェクトのタイプ。

2 この VPA CR が管理するワークロードオブジェクトの名前。

3 モードを **Auto** または **Recreate** に設定します。

- **Auto**:VPA は、Pod の作成時にリソース要求を割り当て、要求されるリソースが新規の推奨事項と大きく異なる場合に、それらを終了して既存の Pod を更新します。
- **Recreate**:VPA は、Pod の作成時にリソース要求を割り当て、要求されるリソースが新規の推奨事項と大きく異なる場合に、それらを終了して既存の Pod を更新します。このモードはほとんど使用されることはありません。リソース要求が変更される際に Pod が再起動されていることを確認する必要がある場合にのみ使用します。



注記

VPA が推奨リソースを判別し、新規 Pod に推奨事項を割り当てる前に、プロジェクトに動作中の Pod がなければなりません。

2.5.3.2. Pod 作成時における VPA 推奨の自動適用

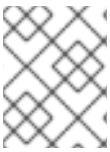
VPA を使用して、Pod が最初にデプロイされる場合にのみ推奨リソースを適用するには、**updateMode** が **Initial** に設定された特定のワークロードオブジェクトの VPA CR を作成します。

次に、VPA の推奨値を使用する必要のあるワークロードオブジェクトに関連付けられた Pod を手動で削除します。**Initial** モードで、VPA は新しいリソースの推奨内容を確認する際に Pod を削除したり、更新したりしません。

Initial モードの VPA CR の例

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment (1)
    name:       frontend (2)
  updatePolicy:
    updateMode: "Initial" (3)
```

1. この VPA CR が管理するワークロードオブジェクトのタイプ。
2. この VPA CR が管理するワークロードオブジェクトの名前。
3. モードを **Initial** に設定します。VPA は、Pod の作成時にリソースを割り当て、Pod の有効期間中はリソースを変更しません。



注記

VPA が推奨リソースを判別し、新規 Pod に推奨事項を割り当てる前に、プロジェクトに動作中の Pod がなければなりません。

2.5.3.3. VPA の推奨事項の手動適用

CPU およびメモリーの推奨値を判別するためだけに VPA を使用するには、**updateMode** を **off** に設定した特定のワークロードオブジェクトの VPA CR を作成します。

Pod がワークロードオブジェクト用に作成されると、VPA はコンテナーの CPU およびメモリーのニーズを分析し、VPA CR の **status** フィールドにそれらの推奨事項を記録します。VPA は、新しい推奨リソースを判別する際に Pod を更新しません。

Off モードの VPA CR の例

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
```

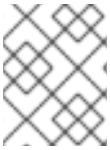
```
metadata:
  name: vpa-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment (1)
    name:       frontend (2)
  updatePolicy:
    updateMode: "Off" (3)
```

1. この VPA CR が管理するワークロードオブジェクトのタイプ。
2. この VPA CR が管理するワークロードオブジェクトの名前。
3. モードを **Off** に設定します。

以下のコマンドを使用して、推奨事項を表示できます。

```
$ oc get vpa <vpa-name> --output yaml
```

この推奨事項により、ワークロードオブジェクトを編集して CPU およびメモリー要求を追加し、推奨リソースを使用して Pod を削除および再デプロイできます。



注記

VPA が推奨リソースを判別する前に、プロジェクトに動作中の Pod がなければなりません。

2.5.3.4. VPA の推奨事項をすべてのコンテナーに適用しないようにする

ワークロードオブジェクトに複数のコンテナーがあり、VPA がすべてのコンテナーを評価および実行対象としないようにするには、特定のワークロードオブジェクトの VPA CR を作成し、**resourcePolicy** を追加して特定のコンテナーをオプトアウトします。

VPA が推奨リソースで Pod を更新すると、**resourcePolicy** が設定されたコンテナーは更新されず、VPA は Pod 内のそれらのコンテナーの推奨事項を提示しません。

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment (1)
    name:       frontend (2)
  updatePolicy:
    updateMode: "Auto" (3)
  resourcePolicy: (4)
    containerPolicies:
    - containerName: my-opt-sidecar
      mode: "Off"
```

1. この VPA CR が管理するワークロードオブジェクトのタイプ。
2. この VPA CR が管理するワークロードオブジェクトの名前。
3. モードを **Auto**、**Recreate**、または **Off** に設定します。**Recreate** モードはほとんど使用されることはありません。リソース要求が変更される際に Pod が再起動されていることを確認する必要がある場合にのみ使用します。
4. オプトアウトするコンテナーを指定し、**mode** を **Off** に設定します。

たとえば、Pod には同じリソース要求および制限の 2 つのコンテナーがあります。

```
# ...
spec:
  containers:
  - name: frontend
    resources:
      limits:
        cpu: 1
        memory: 500Mi
      requests:
        cpu: 500m
        memory: 100Mi
  - name: backend
    resources:
      limits:
        cpu: "1"
        memory: 500Mi
      requests:
        cpu: 500m
        memory: 100Mi
# ...
```

backend コンテナーがオプトアウトに設定された VPA CR を起動した後、VPA は Pod を終了し、**frontend** コンテナーのみに適用される推奨リソースで Pod を再作成します。

```
...
spec:
  containers:
    name: frontend
    resources:
      limits:
        cpu: 50m
        memory: 1250Mi
      requests:
        cpu: 25m
        memory: 262144k
...
    name: backend
    resources:
      limits:
        cpu: "1"
        memory: 500Mi
      requests:
```

```
        cpu: 500m
        memory: 100Mi
...
```

2.5.4. Vertical Pod Autoscaler Operator の使用

VPA カスタムリソース (CR) を作成して、Vertical Pod Autoscaler Operator (VPA) を使用できます。CR は、分析すべき Pod を示し、VPA がそれらの Pod について実行するアクションを判別します。

手順

特定のワークロードオブジェクトの VPA CR を作成するには、以下を実行します。

1. スケーリングするワークロードオブジェクトがあるプロジェクトに切り替えます。
   a. VPA CR YAML ファイルを作成します。

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment (1)
    name:       frontend (2)
  updatePolicy:
    updateMode: "Auto" (3)
  resourcePolicy: (4)
    containerPolicies:
    - containerName: my-opt-sidecar
      mode: "Off"
```

(1) この VPA が管理するワークロードオブジェクトのタイプ (**Deployment**、**StatefulSet**、**Job**、**DaemonSet**、**ReplicaSet**、または **ReplicationController**) を指定します。

(2) この VPA が管理する既存のワークロードオブジェクトの名前を指定します。

(3) VPA モードを指定します。

- **auto** は、コントローラーに関連付けられた Pod に推奨リソースを自動的に適用します。VPA は既存の Pod を終了し、推奨されるリソース制限および要求で新規 Pod を作成します。
- **recreate** は、ワークロードオブジェクトに関連付けられた Pod に推奨リソースを自動的に適用します。VPA は既存の Pod を終了し、推奨されるリソース制限および要求で新規 Pod を作成します。**recreate** モードはほとんど使用されることはありません。リソース要求が変更される際に Pod が再起動されていることを確認する必要がある場合にのみ使用します。
- **initial** は、ワークロードオブジェクトに関連付けられた Pod が作成される際に、推奨リソースを自動的に適用します。VPA は、新しい推奨リソースを確認する際に Pod を更新しません。

- **off** は、ワークロードオブジェクトに関連付けられた Pod の推奨リソースのみを生成します。VPA は、新しい推奨リソースを確認する際に Pod を更新しません。また、新規 Pod に推奨事項を適用しません。

4 オプション:オプトアウトするコンテナーを指定し、モードを **Off** に設定します。

b. VPA CR を作成します。

```
$ oc create -f <file-name>.yaml
```

しばらくすると、VPA はワークロードオブジェクトに関連付けられた Pod 内のコンテナーのリソース使用状況を確認します。

以下のコマンドを実行して、VPA の推奨事項を表示できます。

```
$ oc get vpa <vpa-name> --output yaml
```

出力には、以下のような CPU およびメモリー要求の推奨事項が表示されます。

出力例

```
...
status:

...

  recommendation:
    containerRecommendations:
    - containerName: frontend
      lowerBound: 1
        cpu: 25m
        memory: 262144k
      target: 2
        cpu: 25m
        memory: 262144k
      uncappedTarget: 3
        cpu: 25m
        memory: 262144k
      upperBound: 4
        cpu: 262m
        memory: "274357142"
    - containerName: backend
      lowerBound:
        cpu: 12m
        memory: 131072k
      target:
        cpu: 12m
        memory: 131072k
      uncappedTarget:
        cpu: 12m
        memory: 131072k
      upperBound:
        cpu: 476m
```

```
        memory: "498558823"
...
```

1. **lowerBound** は、推奨リソースの最小レベルです。
2. **target** は、推奨リソースのレベルです。
3. **upperBound** は、推奨リソースの最大レベルです。
4. **uncappedTarget** は最新の推奨リソースです。

2.5.5. Vertical Pod Autoscaler Operator のアンインストール

Vertical Pod Autoscaler Operator (VPA) を OpenShift Container Platform クラスターから削除できます。アンインストール後、既存の VPA CR によってすでに変更された Pod のリソース要求は変更されません。新規 Pod は、Vertical Pod Autoscaler Operator による以前の推奨事項ではなく、ワークロードオブジェクトで定義されるリソースを取得します。

注記

oc delete vpa <vpa-name> コマンドを使用して、特定の VPA を削除できます。Vertical Pod Autoscaler のアンインストール時と同じアクションがリソース要求に対して適用されます。

VPA Operator を削除した後、潜在的な問題を回避するために、Operator に関連する他のコンポーネントを削除することをお勧めします。

前提条件

- Vertical Pod Autoscaler Operator がインストールされていること。

手順

1. OpenShift Container Platform Web コンソールで、**Operators → Installed Operators** をクリックします。
2. **openshift-vertical-pod-autoscaler** プロジェクトに切り替えます。
3. **VerticalPodAutoscaler** Operator を検索し、Options メニューをクリックします。**Uninstall Operator** を選択します。
4. ダイアログボックスで、**Uninstall** をクリックします。
5. オプション: 演算子に関連付けられているすべてのオペランドを削除するには、ダイアログボックスで、**Delete all operand instances for this operator**チェックボックスをオンにします。
6. **Uninstall** をクリックします。
7. オプション: OpenShift CLI を使用して VPA コンポーネントを削除します。
   a. VPA の変更用 Webhook 設定を削除します。

```
$ oc delete mutatingwebhookconfigurations/vpa-webhook-config
```

b. VPA カスタムリソースを一覧表示します。

```
$ oc get
verticalpodautoscalercheckpoints.autoscaling.k8s.io,verticalpodautoscalercontrollers.autosc
ling.openshift.io,verticalpodautoscalers.autoscaling.k8s.io -o wide --all-namespaces
```

出力例

```
NAMESPACE      NAME                                                    AGE
my-project     verticalpodautoscalercheckpoint.autoscaling.k8s.io/vpa-recommender-httpd
5m46s

NAMESPACE                      NAME                                                    AGE
openshift-vertical-pod-autoscaler
verticalpodautoscalercontroller.autoscaling.openshift.io/default   11m

NAMESPACE      NAME                                                MODE   CPU    MEM
PROVIDED   AGE
my-project     verticalpodautoscaler.autoscaling.k8s.io/vpa-recommender   Auto   93m
262144k   True       9m15s
```

c. 一覧表示された VPA カスタムリソースを削除します。以下に例を示します。

```
$ oc delete verticalpodautoscalercheckpoint.autoscaling.k8s.io/vpa-recommender-httpd -
n my-project
```

```
$ oc delete verticalpodautoscalercontroller.autoscaling.openshift.io/default -n openshift-
vertical-pod-autoscaler
```

```
$ oc delete verticalpodautoscaler.autoscaling.k8s.io/vpa-recommender -n my-project
```

d. VPA カスタムリソース定義 (CRD) を一覧表示します。

```
$ oc get crd
```

出力例

```
NAME                                                  CREATED AT
 ...
verticalpodautoscalercheckpoints.autoscaling.k8s.io             2022-02-07T14:09:20Z
verticalpodautoscalercontrollers.autoscaling.openshift.io       2022-02-07T14:09:20Z
verticalpodautoscalers.autoscaling.k8s.io                       2022-02-07T14:09:20Z
 ...
```

e. 一覧表示された VPA CRD を削除します。

```
$ oc delete crd verticalpodautoscalercheckpoints.autoscaling.k8s.io
verticalpodautoscalercontrollers.autoscaling.openshift.io
verticalpodautoscalers.autoscaling.k8s.io
```

CRD を削除すると、関連付けられたすべてのリソース、クラスターロール、およびクラスターロールバインデ

CRD を削除すると、関連付けられたロール、クラスターロール、およびロールバインディングが削除されます。ただし、手動で削除する必要のあるクラスターロールがいくつかあります。

f. VPA クラスターロールを一覧表示します。

```
$ oc get clusterrole | grep openshift-vertical-pod-autoscaler
```

出力例

```
openshift-vertical-pod-autoscaler-6896f-admin       2022-02-02T15:29:55Z
openshift-vertical-pod-autoscaler-6896f-edit        2022-02-02T15:29:55Z
openshift-vertical-pod-autoscaler-6896f-view        2022-02-02T15:29:55Z
```

g. 一覧表示された VPA クラスターロールを削除します。以下に例を示します。

```
$ oc delete clusterrole openshift-vertical-pod-autoscaler-6896f-admin openshift-vertical-pod-autoscaler-6896f-edit openshift-vertical-pod-autoscaler-6896f-view
```

h. VPA Operator を削除します。

```
$ oc delete operator/vertical-pod-autoscaler.openshift-vertical-pod-autoscaler
```

2.6. POD への機密性の高いデータの提供

アプリケーションによっては、パスワードやユーザー名など開発者に使用させない秘密情報が必要になります。

管理者として **シークレット** オブジェクトを使用すると、この情報を平文で公開することなく提供することが可能です。

2.6.1. シークレットについて

Secret オブジェクトタイプはパスワード、OpenShift Container Platform クライアント設定ファイル、プライベートソースリポジトリーの認証情報などの機密情報を保持するメカニズムを提供します。シークレットは機密内容を Pod から切り離します。シークレットはボリュームプラグインを使用してコンテナーにマウントすることも、システムが Pod の代わりにシークレットを使用して各種アクションを実行することもできます。

キーのプロパティーには以下が含まれます。

- シークレットデータはその定義とは別に参照できます。
- シークレットデータのボリュームは一時ファイルストレージ機能 (tmpfs) でサポートされ、ノードで保存されることはありません。
- シークレットデータは namespace 内で共有できます。

YAML Secret オブジェクト定義

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
```

```
  namespace: my-namespace
type: Opaque (1)
data: (2)
  username: dmFsdWUtMQ0K (3)
  password: dmFsdWUtMg0KDQo=
stringData: (4)
  hostname: myapp.mydomain.com (5)
```

(1) シークレットにキー名および値の構造を示しています。

(2) **data** フィールドのキーに使用可能な形式については、Kubernetes identifiers glossary の **DNS_SUBDOMAIN** 値のガイドラインに従う必要があります。

(3) **data** マップのキーに関連付けられる値は base64 でエンコーディングされている必要があります。

(4) **stringData** マップのエントリーが base64 に変換され、このエントリーは自動的に **data** マップに移動します。このフィールドは書き込み専用です。この値は **data** フィールドでのみ返されます。

(5) **stringData** マップのキーに関連付けられた値は単純なテキスト文字列で設定されます。

シークレットに依存する Pod を作成する前に、シークレットを作成する必要があります。

シークレットの作成時に以下を実行します。

- シークレットデータでシークレットオブジェクトを作成します。
- Pod のサービスアカウントをシークレットの参照を許可するように更新します。
- シークレットを環境変数またはファイルとして使用する Pod を作成します (**secret** ボリュームを使用)。

2.6.1.1. シークレットの種類

type フィールドの値で、シークレットのキー名と値の構造を指定します。このタイプを使用して、シークレットオブジェクトにユーザー名とキーの配置を実行できます。検証の必要がない場合には、デフォルト設定の **opaque** タイプを使用してください。

以下のタイプから 1 つ指定して、サーバー側で最小限の検証をトリガーし、シークレットデータに固有のキー名が存在することを確認します。

- **kubernetes.io/service-account-token**。サービスアカウントトークンを使用します。
- **kubernetes.io/basic-auth**。Basic 認証で使用します。
- **kubernetes.io/ssh-auth**.SSH キー認証で使用します。
- **kubernetes.io/tls**。TLS 認証局で使用します。

検証が必要ない場合には **type: Opaque** と指定します。これは、シークレットがキー名または値の規則に準拠しないという意味です。**opaque** シークレットでは、任意の値を含む、体系化されていない **key:value** ペアも利用できます。

注記

example.com/my-secret-type などの他の任意のタイプを指定できます。これらのタイプはサーバー側では実行されませんが、シークレットの作成者がその種類のキー/値の要件に従う意図があることを示します。

シークレットのさまざまなタイプの例については、**シークレットの使用** に関連するコードのサンプルを参照してください。

2.6.1.2. シークレットデータキー

シークレットキーは DNS サブドメインになければなりません。

2.6.2. シークレットの作成方法

管理者は、開発者がシークレットに依存する Pod を作成できるよう事前にシークレットを作成しておく必要があります。

シークレットの作成時に以下を実行します。

1. 秘密にしておきたいデータを含む秘密オブジェクトを作成します。各シークレットタイプに必要な特定のデータは、以下のセクションで非表示になります。

   不透明なシークレットを作成する YAML オブジェクトの例

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: test-secret
   type: Opaque (1)
   data: (2)
     username: dmFsdWUtMQ0K
     password: dmFsdWUtMQ0KDQo=
   stringData: (3)
     hostname: myapp.mydomain.com
     secret.properties: |
       property1=valueA
       property2=valueB
   ```

   (1) シークレットのタイプを指定します。

   (2) エンコードされた文字列およびデータを指定します。

   (3) デコードされた文字列およびデータを指定します。

   data フィールドまたは **stringdata** フィールドの両方ではなく、いずれかを使用してください。

2. Pod のサービスアカウントをシークレットを参照するように更新します。

   シークレットを使用するサービスアカウントの YAML

   ```
   apiVersion: v1
   kind: ServiceAccount
   ```

```
 ...
secrets:
- name: test-secret
```

3. シークレットを環境変数またはファイルとして使用する Pod を作成します (**secret** ボリュームを使用)。

シークレットデータと共にボリュームのファイルが設定された Pod の YAML

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
    - name: secret-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "cat /etc/secret-volume/*" ]
      volumeMounts: 1
          - name: secret-volume
            mountPath: /etc/secret-volume 2
            readOnly: true 3
  volumes:
    - name: secret-volume
      secret:
        secretName: test-secret 4
  restartPolicy: Never
```

1. シークレットが必要な各コンテナーに **volumeMounts** フィールドを追加します。
2. シークレットが表示される未使用のディレクトリー名を指定します。シークレットデータマップの各キーは **mountPath** の下にあるファイル名になります。
3. **true** に設定します。true の場合、ドライバーに読み取り専用ボリュームを提供するように指示します。
4. シークレットの名前を指定します。

シークレットデータと共に環境変数が設定された Pod の YAML

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
    - name: secret-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "export" ]
      env:
        - name: TEST_SECRET_USERNAME_ENV_VAR
          valueFrom:
            secretKeyRef: 1
```

```
            name: test-secret
            key: username
  restartPolicy: Never
```

1 シークレットキーを使用する環境変数を指定します。

シークレットデータと環境変数が設定されたビルド設定の YAML

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: secret-example-bc
spec:
  strategy:
    sourceStrategy:
      env:
      - name: TEST_SECRET_USERNAME_ENV_VAR
        valueFrom:
          secretKeyRef: 1
            name: test-secret
            key: username
```

1 シークレットキーを使用する環境変数を指定します。

2.6.2.1. シークレットの作成に関する制限

シークレットを使用するには、Pod がシークレットを参照できる必要があります。シークレットは、以下の 3 つの方法で Pod で使用されます。

- コンテナーの環境変数を事前に設定するために使用される。
- 1 つ以上のコンテナーにマウントされるボリュームのファイルとして使用される。
- Pod のイメージをプルする際に kubelet によって使用される。

ボリュームタイプのシークレットは、ボリュームメカニズムを使用してデータをファイルとしてコンテナーに書き込みます。イメージプルシークレットは、シークレットを namespace のすべての Pod に自動的に挿入するためにサービスアカウントを使用します。

テンプレートにシークレット定義が含まれる場合、テンプレートで指定のシークレットを使用できるようにするには、シークレットのボリュームソースを検証し、指定されるオブジェクト参照が **Secret** オブジェクトを実際に参照していることを確認できる必要があります。そのため、シークレットはこれに依存する Pod の作成前に作成されている必要があります。最も効果的な方法として、サービスアカウントを使用してシークレットを自動的に挿入することができます。

シークレット API オブジェクトは namespace にあります。それらは同じ namespace の Pod によってのみ参照されます。

個々のシークレットは 1MB のサイズに制限されます。これにより、apiserver および kubelet メモリーを使い切るような大規模なシークレットの作成を防ぐことができます。ただし、小規模なシークレットであってもそれらを数多く作成するとメモリーの消費につながります。

2.6.2.2. 不透明なシークレットの作成

管理者は、不透明なシークレットを作成できます。これにより、任意の値を含むことができる非構造化 **key:value** のペアを格納できます。

手順

1. コントロールプレーンノードの YAML ファイルに **Secret** オブジェクトを作成します。以下に例を示します。

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque 1
data:
  username: dXNlci1uYW1l
  password: cGFzc3dvcmQ=
```

1 不透明なシークレットを指定します。

2. 以下のコマンドを使用して **Secret** オブジェクトを作成します。

```
$ oc create -f <filename>.yaml
```

3. Pod でシークレットを使用するには、以下を実行します。

   a. シークレットの作成方法についてセクションに示すように、Pod のサービスアカウントを更新してシークレットを参照します。

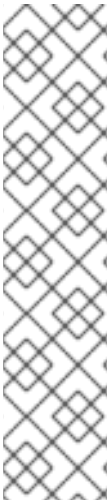
   b. シークレットの作成方法についてに示すように、シークレットを環境変数またはファイル (**secret** ボリュームを使用) として使用する Pod を作成します。

関連情報

- Pod でシークレットを使用する方法は、シークレットの作成方法について を参照してください。

2.6.2.3. サービスアカウントトークンシークレットの作成

管理者は、サービスアカウントトークンシークレットを作成できます。これにより、API に対して認証する必要のあるアプリケーションにサービスアカウントトークンを配布できます。



注記

サービスアカウントトークンシークレットを使用する代わりに、TokenRequest API を使用してバインドされたサービスアカウントトークンを取得することをお勧めします。TokenRequest API から取得したトークンは、有効期間が制限されており、他の API クライアントが読み取れないため、シークレットに保存されているトークンよりも安全です。

TokenRequest API を使用できず、読み取り可能な API オブジェクトで有効期限が切れていないトークンのセキュリティーエクスポージャーが許容できる場合にのみ、サービスアカウントトークンシークレットを作成する必要があります。

バインドされたサービスアカウントトークンの作成に関する詳細は、以下の追加リソースセクションを参照してください。

手順

1. コントロールプレーンノードの YAML ファイルに **Secret** オブジェクトを作成します。

   secret オブジェクトの例:

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-sa-sample
  annotations:
    kubernetes.io/service-account.name: "sa-name" 1
type: kubernetes.io/service-account-token 2
```

   1. 既存のサービスアカウント名を指定します。**ServiceAccount** と **Secret** オブジェクトの両方を作成する場合は、**ServiceAccount** オブジェクトを最初に作成します。
   2. サービスアカウントトークンシークレットを指定します。

2. 以下のコマンドを使用して **Secret** オブジェクトを作成します。

```
$ oc create -f <filename>.yaml
```

3. Pod でシークレットを使用するには、以下を実行します。

   a. シークレットの作成方法についてセクションに示すように、Pod のサービスアカウントを更新してシークレットを参照します。

   b. シークレットの作成方法についてに示すように、シークレットを環境変数またはファイル (**secret** ボリュームを使用) として使用する Pod を作成します。

関連情報

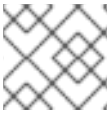
- Pod でシークレットを使用する方法は、シークレットの作成方法について を参照してください。
- バインドされたサービスアカウントトークンの要求については、バインドされたサービスアカウントトークンの使用 を参照してください。

- サービスアカウントの作成については、サービスアカウントの理解と作成 を参照してください。

2.6.2.4. Basic 認証シークレットの作成

管理者は Basic 認証シークレットを作成できます。これにより、Basic 認証に必要な認証情報を保存できます。このシークレットタイプを使用する場合は、**Secret** オブジェクトの **data** パラメーターには、base64 形式でエンコードされた以下のキーが含まれている必要があります。

- **username**: 認証用のユーザー名
- **password**: 認証のパスワードまたはトークン



注記

stringData パラメーターを使用して、クリアテキストコンテンツを使用できます。

手順

1. コントロールプレーンノードの YAML ファイルに **Secret** オブジェクトを作成します。

   secret オブジェクトの例

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-basic-auth
type: kubernetes.io/basic-auth (1)
data:
stringData: (2)
  username: admin
  password: t0p-Secret
```

   (1) Basic 認証のシークレットを指定します。

   (2) 使用する Basic 認証値を指定します。

2. 以下のコマンドを使用して **Secret** オブジェクトを作成します。

```
$ oc create -f <filename>.yaml
```

3. Pod でシークレットを使用するには、以下を実行します。

   a. シークレットの作成方法についてセクションに示すように、Pod のサービスアカウントを更新してシークレットを参照します。

   b. シークレットの作成方法についてに示すように、シークレットを環境変数またはファイル (**secret** ボリュームを使用) として使用する Pod を作成します。

関連情報

- Pod でシークレットを使用する方法は、シークレットの作成方法について を参照してください。

2.6.2.5. SSH 認証シークレットの作成

管理者は、SSH 認証シークレットを作成できます。これにより、SSH 認証に使用されるデータを保存できます。このシークレットタイプを使用する場合、**Secret** オブジェクトの **data** パラメーターには、使用する SSH 認証情報が含まれている必要があります。

手順

1. コントロールプレーンノードの YAML ファイルに **Secret** オブジェクトを作成します。

   secret オブジェクトの例:

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-ssh-auth
type: kubernetes.io/ssh-auth (1)
data:
  ssh-privatekey: | (2)
        MIIEpQIBAAKCAQEAulqb/Y ...
```

   (1) SSH 認証シークレットを指定します。

   (2) SSH のキー/値のペアを、使用する SSH 認証情報として指定します。

2. 以下のコマンドを使用して **Secret** オブジェクトを作成します。

```
$ oc create -f <filename>.yaml
```

3. Pod でシークレットを使用するには、以下を実行します。

   a. シークレットの作成方法についてセクションに示すように、Pod のサービスアカウントを更新してシークレットを参照します。

   b. シークレットの作成方法についてに示すように、シークレットを環境変数またはファイル (**secret** ボリュームを使用) として使用する Pod を作成します。

関連情報

- シークレットの作成方法

2.6.2.6. Docker 設定シークレットの作成

管理者は Docker 設定シークレットを作成できます。これにより、コンテナーイメージレジストリーにアクセスするための認証情報を保存できます。

- **kubernetes.io/dockercfg**.このシークレットタイプを使用してローカルの Docker 設定ファイルを保存します。**secret** オブジェクトの **data** パラメーターには、base64 形式でエンコードされた **.dockercfg** ファイルの内容が含まれている必要があります。

- **kubernetes.io/dockerconfigjson**.このシークレットタイプを使用して、ローカルの Docker 設定 JSON ファイルを保存します。**secret** オブジェクトの **data** パラメーターには、base64 形式でエンコードされた **.docker/config.json** ファイルの内容が含まれている必要があります。

手順

1. コントロールプレーンノードの YAML ファイルに **Secret** オブジェクトを作成します。

Docker 設定の secret オブジェクトの例

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-docker-cfg
  namespace: my-project
type: kubernetes.io/dockerconfig (1)
data:

.dockerconfig:bm5ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2dnZ2cgYXV
0aCBrZXlzCg== (2)
```

(1) シークレットが Docker 設定ファイルを使用することを指定します。

(2) base64 でエンコードされた Docker 設定ファイルの出力

Docker 設定の JSON secret オブジェクトの例

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-docker-json
  namespace: my-project
type: kubernetes.io/dockerconfig (1)
data:

.dockerconfigjson:bm5ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2dnZ2cg
YXV0aCBrZXlzCg== (2)
```

(1) シークレットが Docker 設定の JSON ファイルを使用することを指定します。

(2) base64 でエンコードされた Docker 設定 JSON ファイルの出力

2. 以下のコマンドを使用して **Secret** オブジェクトを作成します。

```
$ oc create -f <filename>.yaml
```

3. Pod でシークレットを使用するには、以下を実行します。

   a. シークレットの作成方法についてセクションに示すように、Pod のサービスアカウントを更新してシークレットを参照します。

   b. シークレットの作成方法についてに示すように、シークレットを環境変数またはファイル (**secret** ボリュームを使用) として使用する Pod を作成します。

関連情報

- Pod でシークレットを使用する方法は、シークレットの作成方法について を参照してください。

2.6.3. シークレットの更新方法

シークレットの値を変更する場合、値 (すでに実行されている Pod で使用される値) は動的に変更されません。シークレットを変更するには、元の Pod を削除してから新規の Pod を作成する必要があります (同じ PodSpec を使用する場合があります)。

シークレットの更新は、新規コンテナーイメージのデプロイメントと同じワークフローで実行されます。**kubectl rolling-update** コマンドを使用できます。

シークレットの **resourceVersion** 値は参照時に指定されません。したがって、シークレットが Pod の起動と同じタイミングで更新される場合、Pod に使用されるシークレットのバージョンは定義されません。

注記

現時点で、Pod の作成時に使用されるシークレットオブジェクトのリソースバージョンを確認することはできません。コントローラーが古い **resourceVersion** を使用して Pod を再起動できるように、Pod がこの情報を報告できるようにすることが予定されています。それまでは既存シークレットのデータを更新せずに別の名前で新規のシークレットを作成します。

2.6.4. シークレットで署名証明書を使用する方法

サービスの通信を保護するため、プロジェクト内のシークレットに追加可能な、署名されたサービス証明書/キーペアを生成するように OpenShift Container Platform を設定することができます。

サービス提供証明書のシークレット は、追加設定なしの証明書を必要とする複雑なミドルウェアアプリケーションをサポートするように設計されています。これにはノードおよびマスターの管理者ツールで生成されるサーバー証明書と同じ設定が含まれます。

サービス提供証明書のシークレット用に設定されるサービス Pod 仕様

```
apiVersion: v1
kind: Service
metadata:
  name: registry
  annotations:
    service.beta.openshift.io/serving-cert-secret-name: registry-cert (1)
# ...
```

(1) 証明書の名前を指定します。

他の Pod は Pod に自動的にマウントされる **/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt** ファイルの CA バンドルを使用して、クラスターで作成される証明書 (内部 DNS 名の場合にのみ署名される) を信頼できます。

この機能の署名アルゴリズムは **x509.SHA256WithRSA** です。ローテーションを手動で実行するには、生成されたシークレットを削除します。新規の証明書が作成されます。

2.6.4.1. シークレットで使用する署名証明書の生成

署名されたサービス証明書/キーペアを Pod で使用するには、サービスを作成または編集して **service.beta.openshift.io/serving-cert-secret-name** アノテーションを追加した後に、シークレットを Pod に追加します。

手順

サービス提供証明書のシークレット を作成するには、以下を実行します。

1. サービスの **Pod** 仕様を編集します。
2. シークレットに使用する名前に **service.beta.openshift.io/serving-cert-secret-name** アノテーションを追加します。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
  annotations:
     service.beta.openshift.io/serving-cert-secret-name: my-cert (1)
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

   証明書およびキーは PEM 形式であり、それぞれ **tls.crt** および **tls.key** に保存されます。

3. サービスを作成します。

```
$ oc create -f <file-name>.yaml
```

4. シークレットを表示して、作成されていることを確認します。
   a. すべてのシークレットの一覧を表示します。

```
$ oc get secrets
```

   出力例

```
NAME                  TYPE                                  DATA      AGE
my-cert               kubernetes.io/tls                     2         9m
```

   b. シークレットの詳細を表示します。

```
$ oc describe secret my-cert
```

   出力例

```
Name:         my-cert
Namespace:    openshift-console
Labels:       <none>
Annotations:  service.beta.openshift.io/expiry: 2023-03-08T23:22:40Z
```

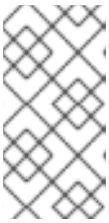
```
            service.beta.openshift.io/originating-service-name: my-service
            service.beta.openshift.io/originating-service-uid: 640f0ec3-afc2-4380-bf31-
a8c784846a11
            service.beta.openshift.io/expiry: 2023-03-08T23:22:40Z

Type:  kubernetes.io/tls

Data
====
tls.key:  1679 bytes
tls.crt:  2595 bytes
```

5. このシークレットを使って **Pod** 仕様を編集します。

```
apiVersion: v1
kind: Pod
metadata:
  name: my-service-pod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
  volumes:
  - name: foo
    secret:
      secretName: my-cert
      items:
      - key: username
        path: my-group/my-username
        mode: 511
```

これが利用可能な場合、Pod が実行されます。この証明書は内部サービス DNS 名、**<service.name>.<service.namespace>.svc** に適しています。

証明書/キーのペアは有効期限に近づくと自動的に置換されます。シークレットの **service.beta.openshift.io/expiry** アノテーションで RFC3339 形式の有効期限の日付を確認します。



注記

ほとんどの場合、サービス DNS 名 **<service.name>.<service.namespace>.svc** は外部にルーティング可能ではありません。**<service.name>.<service.namespace>.svc** の主な使用方法として、クラスターまたはサービス間の通信用として、 re-encrypt ルートで使用されます。

2.6.5. シークレットのトラブルシューティング

サービス証明書の生成は以下を出して失敗します (サービスの **service.beta.openshift.io/serving-cert-generation-error** アノテーションには以下が含まれます)。

```
secret/ssl-key references serviceUID 62ad25ca-d703-11e6-9d6f-0e9c0057b608, which does not
match 77b6dd80-d716-11e6-9d6f-0e9c0057b60
```

-

証明書を生成したサービスがすでに存在しないか、またはサービスに異なる **serviceUID** があります。古いシークレットを削除し、サービスのアノテーション (**service.beta.openshift.io/serving-cert-generation-error**、**service.beta.openshift.io/serving-cert-generation-error-num**) をクリアして証明書の再生成を強制的に実行する必要があります。

1. シークレットを削除します。

```
$ oc delete secret <secret_name>
```

2. アノテーションをクリアします。

```
$ oc annotate service <service_name> service.beta.openshift.io/serving-cert-generation-error-
```

```
$ oc annotate service <service_name> service.beta.openshift.io/serving-cert-generation-error-num-
```

注記

アノテーションを削除するコマンドでは、削除するアノテーション名の後に **-** を付けます。

2.7. 設定マップの作成および使用

以下のセクションでは、設定マップおよびそれらを作成し、使用する方法を定義します。

2.7.1. 設定マップについて

数多くのアプリケーションには、設定ファイル、コマンドライン引数、および環境変数の組み合わせを使用した設定が必要です。OpenShift Container Platform では、これらの設定アーティファクトは、コンテナー化されたアプリケーションを移植可能な状態に保つためにイメージコンテンツから切り離されます。

ConfigMap オブジェクトは、コンテナーを OpenShift Container Platform に依存させないようにする一方で、コンテナーに設定データを挿入するメカニズムを提供します。設定マップは、個々のプロパティーなどの粒度の細かい情報や、設定ファイル全体または JSON Blob などの粒度の荒い情報を保存するために使用できます。

ConfigMap API オブジェクトは、Pod で使用したり、コントローラーなどのシステムコンポーネントの設定データを保存するために使用できる設定データのキーと値のペアを保持します。以下に例を示します。

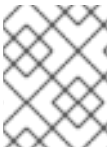
ConfigMap オブジェクト定義

```
kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: default
data: 1
  example.property.1: hello
```

```
  example.property.2: world
  example.property.file: |-
    property.1=value-1
    property.2=value-2
    property.3=value-3
binaryData:
  bar: L3Jvb3QvMTAw 2
```

1 1 設定データが含まれます。

2 バイナリー Java キーストアファイルなどの UTF8 以外のデータを含むファイルを参照します。Base 64 のファイルデータを入力します。



注記

イメージなどのバイナリーファイルから設定マップを作成する場合に、**binaryData** フィールドを使用できます。

設定データはさまざまな方法で Pod 内で使用できます。設定マップは以下を実行するために使用できます。

- コンテナーへの環境変数値の設定
- コンテナーのコマンドライン引数の設定
- ボリュームの設定ファイルの設定

ユーザーとシステムコンポーネントの両方が設定データを設定マップに保存できます。

設定マップはシークレットに似ていますが、機密情報を含まない文字列の使用をより効果的にサポートするように設計されています。

設定マップの制限
設定マップは、コンテンツを Pod で使用される前に作成する必要があります。

コントローラーは、設定データが不足していても、その状況を許容して作成できます。ケースごとに設定マップを使用して設定される個々のコンポーネントを参照してください。

ConfigMap オブジェクトはプロジェクト内にあります。

それらは同じプロジェクトの Pod によってのみ参照されます。

Kubelet は、API サーバーから取得する Pod の設定マップの使用のみをサポートします。

これには、CLI を使用して作成された Pod、またはレプリケーションコントローラーから間接的に作成された Pod が含まれます。これには、OpenShift Container Platform ノードの **--manifest-url** フラグ、その **--config** フラグ、またはその REST API を使用して作成された Pod は含まれません (これらは Pod を作成する一般的な方法ではありません)。

2.7.2. OpenShift Container Platform Web コンソールでの設定マップの作成

OpenShift Container Platform Web コンソールで設定マップを作成できます。

手順

- クラスター管理者として設定マップを作成するには、以下を実行します。
  1. Administrator パースペクティブで **Workloads → Config Maps** を選択します。
  2. ページの右上にある **Create Config Map** を選択します。
  3. 設定マップの内容を入力します。
  4. **Create** を選択します。
- 開発者として設定マップを作成するには、以下を実行します。
  1. 開発者パースペクティブで、**Config Maps** を選択します。
  2. ページの右上にある **Create Config Map** を選択します。
  3. 設定マップの内容を入力します。
  4. **Create** を選択します。

2.7.3. CLI を使用して設定マップを作成する

以下のコマンドを使用して、ディレクトリー、特定のファイルまたはリテラル値から設定マップを作成できます。

手順

- 設定マップの作成

```
$ oc create configmap <configmap_name> [options]
```

2.7.3.1. ディレクトリーからの設定マップの作成

ディレクトリーから設定マップを作成できます。この方法では、ディレクトリー内の複数のファイルを使用して設定マップを作成できます。

手順

以下の例の手順は、ディレクトリーから設定マップを作成する方法を説明しています。

1. 設定マップの設定に必要なデータがすでに含まれるファイルのあるディレクトリーについて見てみましょう。

```
$ ls example-files
```

出力例

```
game.properties
ui.properties
```

```
$ cat example-files/game.properties
```

出力例

```
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
```

```
$ cat example-files/ui.properties
```

出力例

```
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

2. 次のコマンドを入力して、このディレクトリー内の各ファイルの内容を保持する設定マップを作成します。

```
$ oc create configmap game-config \
    --from-file=example-files/
```

--from-file オプションがディレクトリーを参照する場合、そのディレクトリーに直接含まれる各ファイルが ConfigMap でキーを設定するために使用されます。 このキーの名前はファイル名であり、キーの値はファイルの内容になります。

たとえば、前のコマンドは次の設定マップを作成します。

```
$ oc describe configmaps game-config
```

出力例

```
Name:           game-config
Namespace:      default
Labels:         <none>
Annotations:    <none>

Data

game.properties:        158 bytes
ui.properties:          83 bytes
```

マップにある 2 つのキーが、コマンドで指定されたディレクトリーのファイル名に基づいて作成されていることに気づかれることでしょう。それらのキーの内容のサイズは大きくなる可能性があるため、**oc describe** の出力はキーの名前とキーのサイズのみを表示します。

3. **-o** オプションを使用してオブジェクトの **oc get** コマンドを入力し、キーの値を表示します。

```
$ oc get configmaps game-config -o yaml
```

出力例

```
apiVersion: v1
data:
  game.properties: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:34:05Z
  name: game-config
  namespace: default
  resourceVersion: "407"
  selflink: /api/v1/namespaces/default/configmaps/game-config
  uid: 30944725-d66e-11e5-8cd0-68f728db1985
```

2.7.3.2. ファイルから設定マップを作成する

ファイルから設定マップを作成できます。

手順

以下の手順例では、ファイルから設定マップを作成する方法を説明します。

注記

ファイルから設定マップを作成する場合、UTF8 以外のデータを破損することなく、UTF8 以外のデータを含むファイルをこの新規フィールドに配置できます。OpenShift Container Platform はバイナリーファイルを検出し、ファイルを **MIME** として透過的にエンコーディングします。サーバーでは、データを破損することなく **MIME** ペイロードがデコーディングされ、保存されます。

--from-file オプションを CLI に複数回渡すことができます。以下の例を実行すると、ディレクトリーからの作成の例と同等の結果を出すことができます。

1. 特定のファイルを指定して設定マップを作成します。

```
$ oc create configmap game-config-2 \
    --from-file=example-files/game.properties \
    --from-file=example-files/ui.properties
```

2. 結果を確認します。

```
$ oc get configmaps game-config-2 -o yaml
```

出力例

```
apiVersion: v1
data:
  game.properties: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:52:05Z
  name: game-config-2
  namespace: default
  resourceVersion: "516"
  selflink: /api/v1/namespaces/default/configmaps/game-config-2
  uid: b4952dc3-d670-11e5-8cd0-68f728db1985
```

ファイルからインポートされたコンテンツの設定マップで設定するキーを指定できます。これは、**key=value** 式を **--from-file** オプションに渡すことで設定できます。以下に例を示します。

1. キーと値のペアを指定して、設定マップを作成します。

```
$ oc create configmap game-config-3 \
    --from-file=game-special-key=example-files/game.properties
```

2. 結果を確認します。

```
$ oc get configmaps game-config-3 -o yaml
```

出力例

```
apiVersion: v1
data:
  game-special-key: |- (1)
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
```

```
  resourceVersion: "530"
  selflink: /api/v1/namespaces/default/configmaps/game-config-3
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985
```

1 これは、先の手順で設定したキーです。

2.7.3.3. リテラル値からの設定マップの作成

設定マップにリテラル値を指定することができます。

手順

--from-literal オプションは、リテラル値をコマンドラインに直接指定できる **key=value** 構文を取ります。

1. リテラル値を指定して設定マップを作成します。

```
$ oc create configmap special-config \
    --from-literal=special.how=very \
    --from-literal=special.type=charm
```

2. 結果を確認します。

```
$ oc get configmaps special-config -o yaml
```

出力例

```
apiVersion: v1
data:
  special.how: very
  special.type: charm
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: special-config
  namespace: default
  resourceVersion: "651"
  selflink: /api/v1/namespaces/default/configmaps/special-config
  uid: dadce046-d673-11e5-8cd0-68f728db1985
```

2.7.4. ユースケース: Pod で設定マップを使用する

以下のセクションでは、Pod で **ConfigMap** オブジェクトを使用する際のいくつかのユースケースについて説明します。

2.7.4.1. 設定マップの使用によるコンテナーでの環境変数の設定

設定マップはコンテナーで個別の環境変数を設定するために使用したり、有効な環境変数名を生成するすべてのキーを使用してコンテナーで環境変数を設定するために使用したりすることができます。

例として、以下の設定マップについて見てみましょう。

2 つの環境変数を含む ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config 1
  namespace: default 2
data:
  special.how: very 3
  special.type: charm 4
```

1 設定マップの名前。

2 設定マップが存在するプロジェクト。設定マップは同じプロジェクトの Pod によってのみ参照されます。

3 4 挿入する環境変数。

1つの環境変数を含む ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config 1
  namespace: default
data:
  log_level: INFO 2
```

1 設定マップの名前。

2 挿入する環境変数。

手順

- **configMapKeyRef** セクションを使用して、Pod のこの **ConfigMap** のキーを使用できます。

  特定の環境変数を挿入するように設定されている Pod 仕様のサンプル

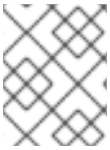
```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env: 1
        - name: SPECIAL_LEVEL_KEY 2
          valueFrom:
            configMapKeyRef:
              name: special-config 3
              key: special.how 4
```

```
      - name: SPECIAL_TYPE_KEY
        valueFrom:
          configMapKeyRef:
            name: special-config 5
            key: special.type 6
            optional: true 7
    envFrom: 8
      - configMapRef:
          name: env-config 9
  restartPolicy: Never
```

1. **ConfigMap** から指定された環境変数をプルするためのスタンザです。
2. キーの値を挿入する Pod 環境変数の名前です。
3. 5. 特定の環境変数のプルに使用する **ConfigMap** の名前です。
4. 6. **ConfigMap** からプルする環境変数です。
7. 環境変数をオプションにします。オプションとして、Pod は指定された **ConfigMap** およびキーが存在しない場合でも起動します。
8. **ConfigMap** からすべての環境変数をプルするためのスタンザです。
9. すべての環境変数のプルに使用する **ConfigMap** の名前です。

この Pod が実行されると、Pod のログには以下の出力が含まれます。

```
SPECIAL_LEVEL_KEY=very
log_level=INFO
```



注記

SPECIAL_TYPE_KEY=charm は出力例に一覧表示されません。**optional: true** が設定されているためです。

2.7.4.2. 設定マップを使用したコンテナーコマンドのコマンドライン引数の設定

設定マップを使用して、コンテナー内のコマンドまたは引数の値を設定することもできます。これは、Kubernetes 置換構文 **$(VAR_NAME)** を使用して実行できます。次の設定マップを検討してください。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

手順

- 値をコンテナーのコマンドに挿入するには、環境変数で ConfigMap を使用する場合のように環境変数として使用する必要のあるキーを使用する必要があります。次に、**$(VAR_NAME)** 構文を使用してコンテナーのコマンドでそれらを参照することができます。

  特定の環境変数を挿入するように設定されている Pod 仕様のサンプル

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)" ]
(1)
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
  restartPolicy: Never
```

(1) 環境変数として使用するキーを使用して、コンテナーのコマンドに値を挿入します。

この Pod が実行されると、test-container コンテナーで実行される echo コマンドの出力は以下のようになります。

```
very charm
```

2.7.4.3. 設定マップの使用によるボリュームへのコンテンツの挿入

設定マップを使用して、コンテンツをボリュームに挿入することができます。

ConfigMap カスタムリソース (CR) の例

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

手順

設定マップを使用してコンテンツをボリュームに挿入するには、2 つの異なるオプションを使用できます。

- 設定マップを使用してコンテンツをボリュームに挿入するための最も基本的な方法は、キーがファイル名であり、ファイルの内容がキーの値になっているファイルでボリュームを設定する方法です。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "cat", "/etc/config/special.how" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config 1
  restartPolicy: Never
```

1 キーを含むファイル。

この Pod が実行されると、cat コマンドの出力は以下のようになります。

```
very
```

- 設定マップキーが投影されるボリューム内のパスを制御することもできます。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "cat", "/etc/config/path/to/special-key" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
        - key: special.how
          path: path/to/special-key 1
  restartPolicy: Never
```

1 設定マップキーへのパス。

この Pod が実行されると、cat コマンドの出力は以下のようになります。

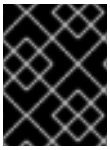
```
very
```

2.8. POD で外部リソースにアクセスするためのデバイスプラグインの使用

デバイスプラグインを使用すると、カスタムコードを作成せずに特定のデバイスタイプ (GPU、InfiniBand、またはベンダー固有の初期化およびセットアップを必要とする他の同様のコンピューティングリソース) を OpenShift Container Platform Pod で使用できます。

2.8.1. デバイスプラグインについて

デバイスプラグインは、クラスター間でハードウェアデバイスを使用する際の一貫した移植可能なソリューションを提供します。デバイスプラグインは、拡張メカニズムを通じてこれらのデバイスをサポートし (これにより、コンテナーがこれらのデバイスを利用できるようになります)、デバイスのヘルスチェックを実施し、それらを安全に共有します。



重要

OpenShift Container Platform はデバイスのプラグイン API をサポートしますが、デバイスプラグインコンテナーは個別のベンダーによりサポートされます。

デバイスプラグインは、特定のハードウェアリソースの管理を行う、ノード上で実行される gRPC サービスです (**kubelet** の外部にあります)。デバイスプラグインは以下のリモートプロシージャーコール (RPC) をサポートしている必要があります。

```
service DevicePlugin {
    // GetDevicePluginOptions returns options to be communicated with Device
    // Manager
    rpc GetDevicePluginOptions(Empty) returns (DevicePluginOptions) {}

    // ListAndWatch returns a stream of List of Devices
    // Whenever a Device state change or a Device disappears, ListAndWatch
    // returns the new list
    rpc ListAndWatch(Empty) returns (stream ListAndWatchResponse) {}

    // Allocate is called during container creation so that the Device
    // Plug-in can run device specific operations and instruct Kubelet
    // of the steps to make the Device available in the container
    rpc Allocate(AllocateRequest) returns (AllocateResponse) {}

    // PreStartcontainer is called, if indicated by Device Plug-in during
    // registration phase, before each container start. Device plug-in
    // can run device specific operations such as reseting the device
    // before making devices available to the container
    rpc PreStartcontainer(PreStartcontainerRequest) returns (PreStartcontainerResponse) {}
}
```

デバイスプラグインの例

- COS ベースのオペレーティングシステム用の Nvidia GPU デバイスプラグイン

- Nvidia の公式 GPU デバイスプラグイン
- Solarflare デバイスプラグイン
- KubeVirt デバイスプラグイン: vfio および kvm

注記

デバイスプラグイン参照の実装を容易にするため
に、**vendor/k8s.io/kubernetes/pkg/kubelet/cm/deviceplugin/device_plugin_stub.go** という Device Manager コードのスタブデバイスプラグインを使用できます。

2.8.1.1. デバイスプラグインのデプロイ方法

- デーモンセットは、デバイスプラグインのデプロイメントに推奨される方法です。
- 起動時にデバイスプラグインは、デバイスマネージャーから RPC を送信するためにノードの **/var/lib/kubelet/device-plugin/** での UNIX ドメインソケットの作成を試行します。
- デバイスプラグインは、ソケットの作成のほかにもハードウェアリソース、ホストファイルシステムへのアクセスを管理する必要があるため、特権付きセキュリティーコンテキストで実行される必要があります。
- デプロイメント手順の詳細については、それぞれのデバイスプラグインの実装で確認できます。

2.8.2. デバイスマネージャーについて

デバイスマネージャーは、特殊なノードのハードウェアリソースを、デバイスプラグインとして知られるプラグインを使って公開するメカニズムを提供します。

特殊なハードウェアは、アップストリームのコード変更なしに公開できます。

重要

OpenShift Container Platform はデバイスのプラグイン API をサポートしますが、デバイスプラグインコンテナーは個別のベンダーによりサポートされます。

デバイスマネージャーはデバイスを **拡張リソース** として公開します。ユーザー Pod は、他の **拡張リソース** を要求するために使用されるのと同じ **制限/要求** メカニズムを使用してデバイスマネージャーで公開されるデバイスを消費できます。

使用開始時に、デバイスプラグインは **/var/lib/kubelet/device-plugins/kubelet.sock** の **Register** を起動してデバイスマネージャーに自己登録し、デバイスマネージャーの要求を提供するために **/var/lib/kubelet/device-plugins/<plugin>.sock** で gRPC サービスを起動します。

デバイスマネージャーは、新規登録要求の処理時にデバイスプラグインサービスで **ListAndWatch** リモートプロシージャーコール (RPC) を起動します。応答としてデバイスマネージャーは gRPC ストリームでプラグインから **デバイス** オブジェクトの一覧を取得します。デバイスマネージャーはプラグインからの新規の更新の有無についてストリームを監視します。プラグイン側では、プラグインはストリームを開いた状態にし、デバイスの状態に変更があった場合には常に新規デバイスの一覧が同じストリーム接続でデバイスマネージャーに送信されます。

新規 Pod の受付要求の処理時に、Kubelet はデバイスの割り当てのために要求された **Extended Resource** をデバイスマネージャーに送信します。デバイスマネージャーはそのデータベースにチェッ

クインして対応するプラグインが存在するかどうかを確認します。プラグインが存在し、ローカルキャッシュと共に割り当て可能な空きデバイスがある場合、**Allocate** RPC がその特定デバイスのプラグインで起動します。

さらにデバイスプラグインは、ドライバーのインストール、デバイスの初期化、およびデバイスのリセットなどの他のいくつかのデバイス固有の操作も実行できます。これらの機能は実装ごとに異なります。

2.8.3. デバイスマネージャーの有効化

デバイスマネージャーを有効にし、デバイスプラグインを実装してアップストリームのコード変更なしに特殊なハードウェアを公開できるようにします。

デバイスマネージャーは、特殊なノードのハードウェアリソースを、デバイスプラグインとして知られるプラグインを使って公開するメカニズムを提供します。

1. 設定するノードタイプの静的な **MachineConfigPool** CRD に関連付けられたラベルを取得します。以下のいずれかの手順を実行します。

   a. マシン設定を表示します。

   ```
   # oc describe machineconfig <name>
   ```

   以下に例を示します。

   ```
   # oc describe machineconfig 00-worker
   ```

   出力例

   ```
   Name:         00-worker
   Namespace:
   Labels:       machineconfiguration.openshift.io/role=worker 1
   ```

   1 デバイスマネージャーに必要なラベル。

手順

1. 設定変更のためのカスタムリソース (CR) を作成します。

   Device Manager CR の設定例

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: KubeletConfig
   metadata:
     name: devicemgr 1
   spec:
     machineConfigPoolSelector:
       matchLabels:
         machineconfiguration.openshift.io: devicemgr 2
     kubeletConfig:
       feature-gates:
         - DevicePlugins=true 3
   ```

1. CR に名前を割り当てます。
2. Machine Config Pool からラベルを入力します。
3. **DevicePlugins** を 'true` に設定します。

2. デバイスマネージャーを作成します。

```
$ oc create -f devicemgr.yaml
```

出力例

```
kubeletconfig.machineconfiguration.openshift.io/devicemgr created
```

3. デバイスマネージャーが実際に有効にされるように、**/var/lib/kubelet/device-plugins/kubelet.sock** がノードで作成されていることを確認します。これは、デバイスマネージャーの gRPC サーバーが新規プラグインの登録がないかどうかリッスンする UNIX ドメインソケットです。このソケットファイルは、デバイスマネージャーが有効にされている場合にのみ Kubelet の起動時に作成されます。

2.9. POD スケジューリングの決定に POD の優先順位を含める

クラスターで Pod の優先順位およびプリエンプションを有効にできます。Pod の優先度は、他の Pod との比較した Pod の重要度を示し、その優先度に基づいて Pod をキューに入れます。Pod のプリエンプションは、クラスターが優先順位の低い Pod のエビクトまたはプリエンプションを実行することを可能にするため、適切なノードに利用可能な領域がない場合に優先順位のより高い Pod をスケジュールできます。 Pod の優先順位は Pod のスケジューリングの順序にも影響を与え、リソース不足の場合のノード上でのエビクションの順序に影響を与えます。

優先順位およびプリエンプションを使用するには、Pod の相対的な重みを定義する優先順位クラスを作成します。次に Pod 仕様で優先順位クラスを参照し、スケジューリングの重みを適用します。

2.9.1. Pod の優先順位について

Pod の優先順位およびプリエンプション機能を使用する場合、スケジューラーは優先順位に基づいて保留中の Pod を順序付け、保留中の Pod はスケジューリングのキューで優先順位のより低い他の保留中の Pod よりも前に置かれます。その結果、より優先順位の高い Pod は、スケジューリングの要件を満たす場合に優先順位の低い Pod よりも早くスケジュールされる可能性があります。Pod をスケジュールできない場合、スケジューラーは引き続き他の優先順位の低い Pod をスケジュールします。

2.9.1.1. Pod の優先順位クラス

Pod には優先順位クラスを割り当てることができます。これは、名前から優先順位の整数値へのマッピングを定義する namespace を使用していないオブジェクトです。値が高いと優先順位が高くなります。

優先順位およびプリエンプションは、1000000000 (10 億) 以下の 32 ビットの整数値を取ることができます。プリエンプションやエビクションを実行すべきでない Critical Pod 用に 10 億より大きい数を予約します。デフォルトで、OpenShift Container Platform には 2 つの予約された優先順位クラスがあり、これらは重要なシステム Pod で保証されたスケジューリングが適用されるために使用されます。

```
$ oc get priorityclasses
```

出力例

```
NAME                     VALUE        GLOBAL-DEFAULT   AGE
cluster-logging          1000000      false            29s
system-cluster-critical   2000000000   false           72m
system-node-critical      2000001000   false           72m
```

- **system-node-critical**: この優先順位クラスには 2000001000 の値があり、ノードからエビクトすべきでないすべての Pod に使用されます。この優先順位クラスを持つ Pod の例として、**sdn-ovs**、**sdn** などがあります。数多くの重要なコンポーネントには、デフォルトで **system-node-critical** の優先順位クラスが含まれます。以下は例になります。
  - master-api
  - master-controller
  - master-etcd
  - sdn
  - sdn-ovs
  - sync
- **system-cluster-critical**: この優先順位クラスには 2000000000 (20 億) の値があり、クラスターに重要な Pod に使用されます。この優先順位クラスの Pod は特定の状況でノードからエビクトされる可能性があります。たとえば、**system-node-critical** 優先順位クラスで設定される Pod が優先される可能性があります。この場合でも、この優先順位クラスではスケジューリングが保証されます。この優先順位クラスを持つ可能性のある Pod の例として、fluentd、descheduler などのアドオンコンポーネントなどがあります。数多くの重要なコンポーネントには、デフォルトで **system-cluster-critical** 優先順位クラスが含まれます。 以下はその一例です。
  - fluentd
  - metrics-server
  - descheduler
- **cluster-logging**: この優先順位は、Fluentd Pod が他のアプリケーションより優先してノードにスケジュールされるようにするために Fluentd で使用されます。

2.9.1.2. Pod の優先順位名

1 つ以上の優先順位クラスを準備した後に、**Pod** 仕様に優先順位クラス名を指定する Pod を作成できます。優先順位の受付コントローラーは、優先順位クラス名フィールドを使用して優先順位の整数値を設定します。名前付きの優先順位クラスが見つからない場合、Pod は拒否されます。

2.9.2. Pod のプリエンプションについて

開発者が Pod を作成する場合、Pod はキューに入れられます。開発者が Pod の優先順位またはプリエンプションを設定している場合、スケジューラーはキューから Pod を選択し、Pod をノードにスケジュールしようとします。スケジューラーが Pod について指定されたすべての要件を満たす適切なノードに領域を見つけられない場合、プリエンプションロジックが保留中の Pod についてトリガーされます。

スケジューラーがノードで 1 つ以上の Pod のプリエンプションを実行する場合、優先順位の高い **Pod** 仕様の **nominatedNodeName** フィールドは、**nodename** フィールドと共にノードの名前に設定されます。スケジューラーは **nominatedNodeName** フィールドを使用して Pod の予約されたリソースを追跡し、またクラスターのプリエンプションについての情報をユーザーに提供します。

スケジューラーが優先順位の低い Pod のプリエンプションを実行した後に、スケジューラーは Pod の正常な終了期間を許可します。スケジューラーが優先順位の低い Pod の終了を待機する間に別のノードが利用可能になると、スケジューラーはそのノードに優先順位の高い Pod をスケジュールできます。その結果、**Pod** 仕様の **nominatedNodeName** フィールドおよび **nodeName** フィールドが異なる可能性があります。

さらに、スケジューラーがノード上で Pod のプリエンプションを実行し、終了を待機している場合で、保留中の Pod よりも優先順位の高い Pod をスケジュールする必要がある場合、スケジューラーは代わりに優先順位の高い Pod をスケジュールできます。その場合、スケジューラーは保留中の Pod の **nominatedNodeName** をクリアし、その Pod を他のノードの対象とすることができます。

プリエンプションは、ノードから優先順位の低いすべての Pod を削除する訳ではありません。スケジューラーは、優先順位の低い Pod の一部を削除して保留中の Pod をスケジュールできます。

スケジューラーは、保留中の Pod をノードにスケジュールできる場合にのみ、Pod のプリエンプションを実行するノードを考慮します。

2.9.2.1. プリエンプションを実行しない優先順位クラス (テクノロジープレビュー)

プリエンプションポリシーが **Never** に設定された Pod は優先順位の低い Pod よりも前のスケジューリングキューに置かれますが、他の Pod のプリエンプションを実行することはできません。スケジュールを待機しているプリエンプションを実行しない Pod は、十分なリソースが解放され、これがスケジュールされるまでスケジュールキュー内に留まります。他の Pod などのプリエンプションを実行しない Pod はスケジューラーのバックオフの対象になります。つまり、スケジューラーがこれらの Pod のスケジュールの試行に成功しない場合、低頻度で再試行されるため、優先順位の低い他の Pod をそれらの Pod よりも前にスケジュールできます。

プリエンプションを実行しない Pod については、他の優先順位の高い Pod が依然としてプリエンプションを実行できます。

2.9.2.2. Pod プリエンプションおよび他のスケジューラーの設定

Pod の優先順位およびプリエンプションを有効にする場合、他のスケジューラー設定を考慮します。

Pod の優先順位および Pod の Disruption Budget (停止状態の予算)

Pod の Disruption Budget (停止状態の予算) は一度に稼働している必要のあるレプリカの最小数またはパーセンテージを指定します。Pod の Disruption Budget (停止状態の予算) を指定する場合、OpenShift Container Platform は、 Best Effort レベルで Pod のプリエンプションを実行する際にそれらを適用します。スケジューラーは、Pod の Disruption Budget (停止状態の予算) に違反しない範囲で Pod のプリエンプションを試行します。該当する Pod が見つからない場合には、Pod の Disruption Budget (停止状態の予算) の要件を無視して優先順位の低い Pod のプリエンプションが実行される可能性があります。

Pod の優先順位およびアフィニティー

Pod のアフィニティーは、新規 Pod が同じラベルを持つ他の Pod と同じノードにスケジュールされることを要求します。

保留中の Pod にノード上の 1 つ以上の優先順位の低い Pod との Pod 間のアフィニティーがある場合、スケジューラーはアフィニティーの要件を違反せずに優先順位の低い Pod のプリエンプションを実行することはできません。この場合、スケジューラーは保留中の Pod をスケジュールするための別の

ノードを探します。ただし、スケジューラーが適切なノードを見つけることは保証できず、保留中のPodがスケジュールされない可能性があります。

この状態を防ぐには、優先順位が等しいPodとのPodのアフィニティーの設定を慎重に行ってください。

2.9.2.3. プリエンプションが実行されたPodの正常な終了

Podのプリエンプションの実行中、スケジューラーはPodの正常な終了期間が期限切れになるのを待機します。その後、Podは機能を完了し、終了します。Podがこの期間後も終了しない場合、スケジューラーはPodを強制終了します。 この正常な終了期間により、スケジューラーによるPodのプリエンプションの実行時と保留中のPodのノードへのスケジュール時に時間差が出ます。

この時間差を最小限にするには、優先順位の低いPodの正常な終了期間を短く設定します。

2.9.3. 優先順位およびプリエンプションの設定

Pod 仕様で **priorityClassName** を使用して優先順位クラスオブジェクトを作成し、Podを優先順位に関連付けることで、Podの優先度およびプリエンプションを適用できます。

優先順位クラスオブジェクトのサンプル

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority (1)
value: 1000000 (2)
preemptionPolicy: PreemptLowerPriority (3)
globalDefault: false (4)
description: "This priority class should be used for XYZ service pods only." (5)
```

(1) 優先順位クラスオブジェクトの名前です。

(2) オブジェクトの優先順位の値です。

(3) この優先順位クラスがプリエンプションを実行するか/しないかを示すオプションのフィールドです。プリエンプションポリシーは、デフォルトで **PreemptLowerPriority** に設定されます。これにより、その優先順位クラスのPodはそれよりも優先順位の低いPodのプリエンプションを実行できます。プリエンプションポリシーが **Never** に設定される場合、その優先順位クラスのPodはプリエンプションを実行しません。

(4) この優先順位クラスが優先順位クラス名が指定されない状態でPodに使用されるかどうかを示すオプションのフィールドです。このフィールドはデフォルトで **false** です。**globalDefault** が **true** に設定される1つの優先順位クラスのみがクラスター内に存在できます。**globalDefault:true** が設定された優先順位クラスがない場合、優先順位クラス名が設定されていないPodの優先順位はゼロになります。**globalDefault:true** が設定された優先順位クラスを追加すると、優先順位クラスが追加された後に作成されたPodのみがその影響を受け、これによって既存Podの優先順位は変更されません。

(5) 開発者がこの優先順位クラスで使用する必要のあるPodを記述するオプションのテキスト文字列です。

手順

優先順位およびプリエンプションを使用するようにクラスターを設定するには、以下を実行します。

1. 1 つ以上の優先順位クラスを作成します。
   a. 優先順位の名前および値を指定します。
   b. 優先順位クラスおよび説明に **globalDefault** フィールドをオプションで指定します。
2. **Pod** 仕様を作成するか、または既存の Pod を編集して、以下のように優先順位クラスの名前を含めます。

   優先順位クラス名を持つ Pod 仕様サンプル

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  priorityClassName: high-priority 1
```

   1 この Pod で使用する優先順位クラスを指定します。

3. Pod を作成します。

```
$ oc create -f <file-name>.yaml
```

   優先順位の名前は Pod 設定または Pod テンプレートに直接追加できます。

2.10. ノードセレクターの使用による特定ノードへの POD の配置

ノードセレクター は、キーと値のペアのマップを指定します。ルールは、ノード上のカスタムラベルと Pod で指定されたセレクターを使って定義されます。

Pod がノードで実行する要件を満たすには、Pod はノードのラベルとして示されるキーと値のペアを持っている必要があります。

同じ Pod 設定でノードのアフィニティーとノードセレクターを使用している場合、以下の重要な考慮事項を参照してください。

2.10.1. ノードセレクターの使用による Pod 配置の制御

Pod でノードセレクターを使用し、ノードでラベルを使用して、Pod がスケジュールされる場所を制御できます。ノードセレクターにより、OpenShift Container Platform は一致するラベルが含まれるノード上に Pod をスケジュールします。

ラベルをノード、マシンセット、またはマシン設定に追加します。マシンセットにラベルを追加すると、ノードまたはマシンが停止した場合に、新規ノードにそのラベルが追加されます。ノードまたはマシン設定に追加されるラベルは、ノードまたはマシンが停止すると維持されません。

ノードセレクターを既存 Pod に追加するには、ノードセレクターを **ReplicaSet** オブジェクト、**DaemonSet** オブジェクト、**StatefulSet** オブジェクト、**Deployment** オブジェクト、または **DeploymentConfig** オブジェクトなどの Pod の制御オブジェクトに追加します。制御オブジェクト下の既存 Pod は、一致するラベルを持つノードで再作成されます。新規 Pod を作成する場合、ノードセレクターを **Pod** 仕様に直接追加できます。

注記

ノードセレクターを既存のスケジュールされている Pod に直接追加することはできません。

前提条件

ノードセレクターを既存 Pod に追加するには、Pod の制御オブジェクトを判別します。たとえば、**router-default-66d5cf9464-m2g75** Pod は **router-default-66d5cf9464** レプリカセットによって制御されます。

```
$ oc describe pod router-default-66d5cf9464-7pwkc

Name:             router-default-66d5cf9464-7pwkc
Namespace:        openshift-ingress

....

Controlled By:    ReplicaSet/router-default-66d5cf9464
```

Web コンソールでは、Pod YAML の **ownerReferences** に制御オブジェクトを一覧表示します。

```
ownerReferences:
  - apiVersion: apps/v1
    kind: ReplicaSet
    name: router-default-66d5cf9464
    uid: d81dd094-da26-11e9-a48a-128e7edf0312
    controller: true
    blockOwnerDeletion: true
```

手順

1. マシンセットを使用するか、またはノードを直接編集してラベルをノードに追加します。
   - **MachineSet** オブジェクトを使用して、ノードの作成時にマシンセットによって管理されるノードにラベルを追加します。
     a. 以下のコマンドを実行してラベルを **MachineSet** オブジェクトに追加します。

```
$ oc patch MachineSet <name> --type='json' -
p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"<key>="
<value>","<key>"="<value>"}}]'  -n openshift-machine-api
```

以下に例を示します。

```
$ oc patch MachineSet abc612-msrtw-worker-us-east-1c  --type='json' -
p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
node","region":"east"}}]'  -n openshift-machine-api
```

b. **oc edit** コマンドを使用して、ラベルが **MachineSet** オブジェクトに追加されていることを確認します。
以下に例を示します。

```
$ oc edit MachineSet abc612-msrtw-worker-us-east-1c -n openshift-machine-api
```

MachineSet オブジェクトの例

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet

....

spec:
...
  template:
    metadata:
...
    spec:
      metadata:
        labels:
          region: east
          type: user-node
....
```

- ラベルをノードに直接追加します。

a. ノードの **Node** オブジェクトを編集します。

```
$ oc label nodes <name> <key>=<value>
```

たとえば、ノードにラベルを付けるには、以下を実行します。

```
$ oc label nodes ip-10-0-142-25.ec2.internal type=user-node region=east
```

b. ラベルがノードに追加されていることを確認します。

```
$ oc get nodes -l type=user-node,region=east
```

出力例

```
NAME                          STATUS   ROLES    AGE   VERSION
ip-10-0-142-25.ec2.internal   Ready    worker   17m   v1.18.3+002a51f
```

2. 一致するノードセレクターを Pod に追加します。

- ノードセレクターを既存 Pod および新規 Pod に追加するには、ノードセレクターを Pod の制御オブジェクトに追加します。

ラベルを含む ReplicaSet オブジェクトのサンプル

```
kind: ReplicaSet
```

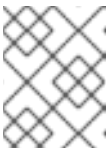
```
....

spec:

....

  template:
    metadata:
      creationTimestamp: null
      labels:
        ingresscontroller.operator.openshift.io/deployment-ingresscontroller: default
        pod-template-hash: 66d5cf9464
    spec:
      nodeSelector:
        kubernetes.io/os: linux
        node-role.kubernetes.io/worker: ''
        type: user-node 1
```

1 ノードセレクターを追加します。

- ノードセレクターを特定の新規 Pod に追加するには、セレクターを **Pod** オブジェクトに直接追加します。

ノードセレクターを持つ Pod オブジェクトの例

```
apiVersion: v1
kind: Pod

....

spec:
  nodeSelector:
    region: east
    type: user-node
```



注記

ノードセレクターを既存のスケジュールされている Pod に直接追加することはできません。

第3章 POD のノードへの配置の制御 (スケジューリング)

3.1. スケジューラーによる POD 配置の制御

Pod のスケジューリングは、クラスター内のノードへの新規 Pod の配置を決定する内部プロセスです。

スケジューラーコードは、新規 Pod の作成時にそれらを確認し、それらをホストするのに最も適したノードを識別します。次に、マスター API を使用して Pod のバインディング (Pod とノードのバインディング) を作成します。

デフォルトの Pod スケジューリング

OpenShift Container Platform には、ほとんどのユーザーのニーズに対応する デフォルトスケジューラー が同梱されます。デフォルトスケジューラーは、Pod に最適なノードを判別するために固有のツールとカスタマイズ可能なツールの両方を使用します。

詳細な Pod スケジューリング

新規 Pod の配置場所に対する制御を強化する必要がある場合、OpenShift Container Platform の詳細スケジューリング機能を使用すると、Pod が特定ノード上か、または特定の Pod と共に実行されることを要求する (または実行されることが優先される) よう Pod を設定することができます。

- Pod のアフィニティーおよび非アフィニティールール の使用。
- Pod のアフィニティー での Pod 配置の制御。
- ノードのアフィニティー での Pod 配置の制御。
- オーバーコミットノード での Pod の配置。
- ノードセレクター での Pod 配置の制御。
- テイントおよび容認 (Toleration) での Pod 配置の制御 。

3.1.1. スケジューラーの使用例

OpenShift Container Platform 内でのスケジューリングの重要な使用例として、柔軟なアフィニティーと非アフィニティーポリシーのサポートを挙げることができます。

3.1.1.1. インフラストラクチャーのトポロジーレベル

管理者は、ノードにラベルを指定することで、インフラストラクチャー (ノード) の複数のトポロジーレベルを定義することができます。たとえば、**region=r1**、**zone=z1**、**rack=s1** などはそれらの例になります。

これらのラベル名には特別な意味はなく、管理者はそれらのインフラストラクチャーラベルに任意の名前 (例: 都市/建物/部屋) を付けることができます。さらに、管理者はインフラストラクチャートポロジーに任意の数のレベルを定義できます。通常は、(**regions → zones → racks**) などの 3 つのレベルが適切なサイズです。管理者はこれらのレベルのそれぞれにアフィニティーと非アフィニティールールを任意の組み合わせで指定することができます。

3.1.1.2. アフィニティー

管理者は、任意のトポロジーレベルまたは複数のレベルでもアフィニティーを指定できるようにスケジューラーを設定することができます。特定レベルのアフィニティーは、同じサービスに属するすべて

の Pod が同じレベルに属するノードにスケジュールされることを示します。これは、管理者がピア Pod が地理的に離れ過ぎないようにすることでアプリケーションの待機時間の要件に対応します。同じアフィニティーグループ内で Pod をホストするために利用できるノードがない場合、Pod はスケジュールされません。

Pod がスケジュールされる場所に対する制御を強化する必要がある場合は、ノードアフィニティールールを使用したノードへの Pod 配置の制御、および アフィニティーおよび非アフィニティールールを使用した他の Pod に応じた Pod 配置 について参照してください。

これらの高度なスケジュール機能を使うと、管理者は Pod をスケジュールするノードを指定でき、他の Pod との比較でスケジューリングを実行したり、拒否したりすることができます。

3.1.1.3. 非アフィニティー

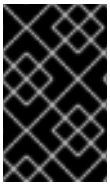
管理者は、任意のトポロジーレベルまたは複数のレベルでも非アフィニティーを設定できるようスケジューラーを設定することができます。特定レベルの非アフィニティー (または分散) は、同じサービスに属するすべての Pod が該当レベルに属するノード全体に分散されることを示します。これにより、アプリケーションが高可用性の目的で適正に分散されます。スケジューラーは、可能な限り均等になるようにすべての適用可能なノード全体にサービス Pod を配置しようとします。

Pod がスケジュールされる場所に対する制御を強化する必要がある場合は、ノードアフィニティールールを使用したノードへの Pod 配置の制御、および アフィニティーおよび非アフィニティールールを使用した他の Pod に応じた Pod 配置 について参照してください。

これらの高度なスケジュール機能を使うと、管理者は Pod をスケジュールするノードを指定でき、他の Pod との比較でスケジューリングを実行したり、拒否したりすることができます。

3.2. デフォルトスケジューラーの設定による POD 配置の制御

OpenShift Container Platform のデフォルトの Pod スケジューラーは、クラスター内のノードにおける新規 Pod の配置場所を判別します。スケジューラーは Pod からのデータを読み取り、設定されるポリシーに基づいて適切なノードを見つけようとします。これは完全に独立した機能であり、スタンドアロン/プラグ可能ソリューションです。Pod を変更することはなく、Pod を特定ノードに関連付ける Pod のバインディングのみを作成します。



重要

スケジューラーポリシーの設定は非推奨となり、今後のリリースで削除される予定です。テクノロジープレビューの代替オプションについての詳細は、スケジューラープロファイルを使用した Pod のスケジューリング を参照してください。

述語と優先順位を選択することで、スケジューラーのポリシーを定義できます。述語と優先順位の一覧は、スケジューラーポリシーの変更 を参照してください。

デフォルトスケジューラーオブジェクトのサンプル

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  annotations:
    release.openshift.io/create-only: "true"
  creationTimestamp: 2019-05-20T15:39:01Z
  generation: 1
  name: cluster
```

```
  resourceVersion: "1491"
  selfLink: /apis/config.openshift.io/v1/schedulers/cluster
  uid: 6435dd99-7b15-11e9-bd48-0aec821b8e34
spec:
  policy: 1
    name: scheduler-policy
  defaultNodeSelector: type=user-node,region=east 2
```

1 カスタムスケジューラーポリシーファイルの名前を指定できます。

2 オプション: Pod の配置を特定のノードに制限するためにデフォルトノードセレクターを指定します。デフォルトのノードセレクターはすべての namespace で作成された Pod に適用されます。Pod は、デフォルトのノードセレクターおよび既存の Pod のノードセレクターに一致するラベルのあるノードにスケジュールできます。このフィールドが設定されている場合でも、プロジェクトスコープのノードセレクターを持つ namespace は影響を受けません。

3.2.1. デフォルトスケジューリングについて

既存の汎用スケジューラーはプラットフォームで提供されるデフォルトのスケジューラー **エンジン** であり、Pod をホストするノードを 3 つの手順で選択します。

ノードのフィルター

利用可能なノードは、指定される制約や要件に基づいてフィルターされます。フィルターは、各ノードで **述語** というフィルター関数の一覧を使用して実行されます。

フィルターされたノード一覧の優先順位付け

優先順位付けは、各ノードに一連の優先度関数を実行することによって行われます。この関数は 0 - 10 までのスコアをノードに割り当て、0 は不適切であることを示し、10 は Pod のホストに適していることを示します。スケジューラー設定は、それぞれの優先度関数について単純な **重み** (正の数値) を取ることができます。各優先度関数で指定されるノードのスコアは重み (ほとんどの優先度のデフォルトの重みは 1) で乗算され、すべての優先度で指定されるそれぞれのノードのスコアを追加して組み合わされます。この重み属性は、一部の優先度により重きを置くようにするなどの目的で管理者によって使用されます。

最適ノードの選択

ノードの並び替えはそれらのスコアに基づいて行われ、最高のスコアを持つノードが Pod をホストするように選択されます。複数のノードに同じ高スコアが付けられている場合、それらのいずれかがランダムに選択されます。

3.2.1.1. スケジューラーポリシーについて

述語と優先順位を選択することで、スケジューラーのポリシーを定義します。

スケジューラー設定ファイルは JSON ファイルであり、**policy.cfg** という名前にする必要があります。これは、スケジューラーが反映する述語と優先順位を指定します。

スケジューラーポリシーがない場合、デフォルトのスケジューラーの動作が使用されます。

重要

スケジューラー設定ファイルで定義される述語および優先度は、デフォルトのスケジューラーポリシーを完全に上書きします。デフォルトの述語および優先順位のいずれかが必要な場合、ポリシーの設定でその関数を明示的に指定する必要があります。

スケジューラー設定マップの例

```
apiVersion: v1
data:
  policy.cfg: |
    {
        "kind" : "Policy",
        "apiVersion" : "v1",
        "predicates" : [
                {"name" : "MaxGCEPDVolumeCount"},
                {"name" : "GeneralPredicates"}, 1
                {"name" : "MaxAzureDiskVolumeCount"},
                {"name" : "MaxCSIVolumeCountPred"},
                {"name" : "CheckVolumeBinding"},
                {"name" : "MaxEBSVolumeCount"},
                {"name" : "MatchInterPodAffinity"},
                {"name" : "CheckNodeUnschedulable"},
                {"name" : "NoDiskConflict"},
                {"name" : "NoVolumeZoneConflict"},
                {"name" : "PodToleratesNodeTaints"}
                ],
        "priorities" : [
                {"name" : "LeastRequestedPriority", "weight" : 1},
                {"name" : "BalancedResourceAllocation", "weight" : 1},
                {"name" : "ServiceSpreadingPriority", "weight" : 1},
                {"name" : "NodePreferAvoidPodsPriority", "weight" : 1},
                {"name" : "NodeAffinityPriority", "weight" : 1},
                {"name" : "TaintTolerationPriority", "weight" : 1},
                {"name" : "ImageLocalityPriority", "weight" : 1},
                {"name" : "SelectorSpreadPriority", "weight" : 1},
                {"name" : "InterPodAffinityPriority", "weight" : 1},
                {"name" : "EqualPriority", "weight" : 1}
                ]
    }
kind: ConfigMap
metadata:
  creationTimestamp: "2019-09-17T08:42:33Z"
  name: scheduler-policy
  namespace: openshift-config
  resourceVersion: "59500"
  selfLink: /api/v1/namespaces/openshift-config/configmaps/scheduler-policy
  uid: 17ee8865-d927-11e9-b213-02d1e1709840`
```

1 **GeneralPredicates** 述語は **PodFitsResources**、**HostName**、**PodFitsHostPorts**、および **MatchNodeSelector** 述語を表します。同じ述語を複数回設定することは許可されていないため、**GeneralPredicates** 述語を、表現される 4 つの述語と共に使用することはできません。

3.2.2. スケジューラーポリシーファイルの作成

デフォルトのスケジューリング動作を変更するには、必要な述語および優先順位を使用して JSON ファイルを作成します。次に、JSON ファイルから設定マップを生成し、設定マップを使用するように **cluster** スケジューラーオブジェクトを指定します。

手順

スケジューラーポリシーを設定するには、以下を実行します。

1. 必要な述語と優先順位を使って **policy.cfg** という名前の JSON ファイルを作成します。

スケジューラー JSON ファイルのサンプル

```
{
"kind" : "Policy",
"apiVersion" : "v1",
"predicates" : [ 1
        {"name" : "MaxGCEPDVolumeCount"},
        {"name" : "GeneralPredicates"},
        {"name" : "MaxAzureDiskVolumeCount"},
        {"name" : "MaxCSIVolumeCountPred"},
        {"name" : "CheckVolumeBinding"},
        {"name" : "MaxEBSVolumeCount"},
        {"name" : "MatchInterPodAffinity"},
        {"name" : "CheckNodeUnschedulable"},
        {"name" : "NoDiskConflict"},
        {"name" : "NoVolumeZoneConflict"},
        {"name" : "PodToleratesNodeTaints"}
        ],
"priorities" : [ 2
        {"name" : "LeastRequestedPriority", "weight" : 1},
        {"name" : "BalancedResourceAllocation", "weight" : 1},
        {"name" : "ServiceSpreadingPriority", "weight" : 1},
        {"name" : "NodePreferAvoidPodsPriority", "weight" : 1},
        {"name" : "NodeAffinityPriority", "weight" : 1},
        {"name" : "TaintTolerationPriority", "weight" : 1},
        {"name" : "ImageLocalityPriority", "weight" : 1},
        {"name" : "SelectorSpreadPriority", "weight" : 1},
        {"name" : "InterPodAffinityPriority", "weight" : 1},
        {"name" : "EqualPriority", "weight" : 1}
        ]
}
```

1. 必要に応じて述語を追加します。
2. 必要に応じて優先順位を追加します。

2. スケジューラー JSON ファイルに基づいて設定マップを作成します。

```
$ oc create configmap -n openshift-config --from-file=policy.cfg <configmap-name> 1
```

1. 設定マップの名前を入力します。

以下に例を示します。

```
$ oc create configmap -n openshift-config --from-file=policy.cfg scheduler-policy
```

出力例

```
configmap/scheduler-policy created
```

3. スケジューラー Operator カスタムリソースを編集して設定マップを追加します。

```
$ oc patch Scheduler cluster --type='merge' -p '{"spec":{"policy":{"name":"<configmap-name>"}}}' --type=merge 1
```

1 設定マップの名前を指定します。

以下に例を示します。

```
$ oc patch Scheduler cluster --type='merge' -p '{"spec":{"policy":{"name":"scheduler-policy"}}}' --type=merge
```

Scheduler 設定リソースに変更を加えた後に、**openshift-kube-apiserver** Pod の再デプロイを待機します。これには数分の時間がかかる場合があります。Pod が再デプロイされるまで、新規スケジューラーは有効になりません。

4. **openshift-kube-scheduler** namespace のスケジューラー Pod のログを表示して、スケジューラーポリシーが設定されていることを確認します。以下のコマンドは、スケジューラーによって登録される述語と優先順位をチェックします。

```
$ oc logs <scheduler-pod> | grep predicates
```

以下に例を示します。

```
$ oc logs openshift-kube-scheduler-ip-10-0-141-29.ec2.internal | grep predicates
```

出力例

```
Creating scheduler with fit predicates 'map[MaxGCEPDVolumeCount:{} MaxAzureDiskVolumeCount:{} CheckNodeUnschedulable:{} NoDiskConflict:{} NoVolumeZoneConflict:{} GeneralPredicates:{} MaxCSIVolumeCountPred:{} CheckVolumeBinding:{} MaxEBSVolumeCount:{} MatchInterPodAffinity:{} PodToleratesNodeTaints:{}]' and priority functions 'map[InterPodAffinityPriority:{} LeastRequestedPriority:{} ServiceSpreadingPriority:{} ImageLocalityPriority:{} SelectorSpreadPriority:{} EqualPriority:{} BalancedResourceAllocation:{} NodePreferAvoidPodsPriority:{} NodeAffinityPriority:{} TaintTolerationPriority:{}]'
```

3.2.3. スケジューラーポリシーの変更

openshift-config プロジェクトでスケジューラーポリシーの設定マップを作成または編集して、スケジューリング動作を変更します。**scheduler policy** を作成するには、述語と優先順位の追加および削除を設定マップに対して実行します。

手順

現在のカスタムスケジュールを変更するには、以下のいずれかの方法を使用します。

- スケジューラーポリシーの設定マップを編集します。

```
$ oc edit configmap <configmap-name>  -n openshift-config
```

以下に例を示します。

```
$ oc edit configmap scheduler-policy -n openshift-config
```

出力例

```
apiVersion: v1
data:
  policy.cfg: |
    {
        "kind" : "Policy",
        "apiVersion" : "v1",
        "predicates" : [ 1
                {"name" : "MaxGCEPDVolumeCount"},
                {"name" : "GeneralPredicates"},
                {"name" : "MaxAzureDiskVolumeCount"},
                {"name" : "MaxCSIVolumeCountPred"},
                {"name" : "CheckVolumeBinding"},
                {"name" : "MaxEBSVolumeCount"},
                {"name" : "MatchInterPodAffinity"},
                {"name" : "CheckNodeUnschedulable"},
                {"name" : "NoDiskConflict"},
                {"name" : "NoVolumeZoneConflict"},
                {"name" : "PodToleratesNodeTaints"}
                ],
        "priorities" : [ 2
                {"name" : "LeastRequestedPriority", "weight" : 1},
                {"name" : "BalancedResourceAllocation", "weight" : 1},
                {"name" : "ServiceSpreadingPriority", "weight" : 1},
                {"name" : "NodePreferAvoidPodsPriority", "weight" : 1},
                {"name" : "NodeAffinityPriority", "weight" : 1},
                {"name" : "TaintTolerationPriority", "weight" : 1},
                {"name" : "ImageLocalityPriority", "weight" : 1},
                {"name" : "SelectorSpreadPriority", "weight" : 1},
                {"name" : "InterPodAffinityPriority", "weight" : 1},
                {"name" : "EqualPriority", "weight" : 1}
                ]
    }
kind: ConfigMap
metadata:
  creationTimestamp: "2019-09-17T17:44:19Z"
  name: scheduler-policy
  namespace: openshift-config
  resourceVersion: "15370"
  selfLink: /api/v1/namespaces/openshift-config/configmaps/scheduler-policy
```

1. 必要に応じて述語を追加または削除します。
2. 必要に応じて述語の重みを追加、削除、または変更します。

スケジューラーが更新されたポリシーで Pod を再起動するまでに数分の時間がかかる場合があります。

- 使用されるポリシーと述語を変更します。

1. スケジューラーポリシーの設定マップを削除します。

```
$ oc delete configmap -n openshift-config <name>
```

以下に例を示します。

```
$ oc delete configmap -n openshift-config  scheduler-policy
```

2. **policy.cfg** ファイルを編集し、必要に応じてポリシーおよび述語を追加し、削除します。以下に例を示します。

```
$ vi policy.cfg
```

出力例

```
apiVersion: v1
data:
  policy.cfg: |
    {
    "kind" : "Policy",
    "apiVersion" : "v1",
    "predicates" : [
            {"name" : "MaxGCEPDVolumeCount"},
            {"name" : "GeneralPredicates"},
            {"name" : "MaxAzureDiskVolumeCount"},
            {"name" : "MaxCSIVolumeCountPred"},
            {"name" : "CheckVolumeBinding"},
            {"name" : "MaxEBSVolumeCount"},
            {"name" : "MatchInterPodAffinity"},
            {"name" : "CheckNodeUnschedulable"},
            {"name" : "NoDiskConflict"},
            {"name" : "NoVolumeZoneConflict"},
            {"name" : "PodToleratesNodeTaints"}
            ],
    "priorities" : [
            {"name" : "LeastRequestedPriority", "weight" : 1},
            {"name" : "BalancedResourceAllocation", "weight" : 1},
            {"name" : "ServiceSpreadingPriority", "weight" : 1},
            {"name" : "NodePreferAvoidPodsPriority", "weight" : 1},
            {"name" : "NodeAffinityPriority", "weight" : 1},
            {"name" : "TaintTolerationPriority", "weight" : 1},
            {"name" : "ImageLocalityPriority", "weight" : 1},
            {"name" : "SelectorSpreadPriority", "weight" : 1},
            {"name" : "InterPodAffinityPriority", "weight" : 1},
            {"name" : "EqualPriority", "weight" : 1}
            ]
    }
```

3. スケジューラー JSON ファイルに基づいてスケジューラーポリシーの設定マップを再作成します。

```
$ oc create configmap -n openshift-config --from-file=policy.cfg <configmap-name> 1
```

1 設定マップの名前を入力します。

以下に例を示します。

```
$ oc create configmap -n openshift-config --from-file=policy.cfg scheduler-policy
```

出力例

```
configmap/scheduler-policy created
```

3.2.3.1. スケジューラーの述語について

述語は、不適切なノードをフィルターに掛けるルールです。

OpenShift Container Platform には、デフォルトでいくつかの述語が提供されています。これらの述語の一部は、特定のパラメーターを指定してカスタマイズできます。複数の述語を組み合わせてノードの追加フィルターを指定できます。

3.2.3.1.1. 静的な述語

これらの述語はユーザーから設定パラメーターまたは入力を取りません。これらはそれぞれの正確な名前を使用してスケジューラー設定に指定されます。

3.2.3.1.1.1. デフォルトの述語

デフォルトのスケジューラーポリシーには以下の述語が含まれます。

NoVolumeZoneConflict 述語は Pod が要求するボリュームがゾーンで利用可能であることを確認します。

```
{"name" : "NoVolumeZoneConflict"}
```

MaxEBSVolumeCount 述語は、AWS インスタンスに割り当てることのできるボリュームの最大数を確認します。

```
{"name" : "MaxEBSVolumeCount"}
```

MaxAzureDiskVolumeCount 述語は Azure ディスクボリュームの最大数をチェックします。

```
{"name" : "MaxAzureDiskVolumeCount"}
```

PodToleratesNodeTaints 述語は Pod がノードテイントを許容できるかどうかをチェックします。

```
{"name" : "PodToleratesNodeTaints"}
```

CheckNodeUnschedulable 述語は、Pod を **Unschedulable** 仕様でノード上にスケジュールできるかどうかをチェックします。

```
{"name" : "CheckNodeUnschedulable"}
```

CheckVolumeBinding 述語は、バインドされた PVC とバインドされていない PVC の両方について Pod が要求するボリュームに基づいて Pod が適切かどうかを評価します。

- バインドされる PVC の場合、述語は対応する PV のノードアフィニティーが指定ノードで満たされていることをチェックします。
- バインドされない PVC の場合、述語は PVC 要件を満たし、PV ノードのアフィニティーが指定ノードで満たされる利用可能な PV を検索します。

述語は、すべてのバインドされる PVC にノードと互換性のある PV がある場合や、すべてのバインドされていない PVC が利用可能なノードと互換性のある PV に一致する場合に true を返します。

```
{"name" : "CheckVolumeBinding"}
```

NoDiskConflict 述語は Pod が要求するボリュームが利用可能であるかどうかを確認します。

```
{"name" : "NoDiskConflict"}
```

MaxGCEPDVolumeCount 述語は、Google Compute Engine (GCE) 永続ディスク (PD) の最大数を確認します。

```
{"name" : "MaxGCEPDVolumeCount"}
```

MaxCSIVolumeCountPred 述語は、ノードに割り当てられる Container Storage Interface (CSI) ボリュームの数と、その数が設定した制限を超えるかどうかを判別します。

```
{"name" : "MaxCSIVolumeCountPred"}
```

MatchInterPodAffinity 述語は、Pod のアフィニティー/非アフィニティールールが Pod を許可するかどうかを確認します。

```
{"name" : "MatchInterPodAffinity"}
```

3.2.3.1.1.2. 他の静的な述語

OpenShift Container Platform は以下の述語もサポートしています。

注記

CheckNode-* 述語は、Taint Nodes By Condition 機能が有効にされている場合は使用できません。Taint Nodes By Condition 機能はデフォルトで有効にされています。

CheckNodeCondition 述語は、**out of disk (ディスク不足)**、**network unavailable (ネットワークが使用不可)**、または **not ready (準備できていない)** 状態を報告するノードで Pod をスケジュールできるかどうかを確認します。

```
{"name" : "CheckNodeCondition"}
```

CheckNodeLabelPresence 述語は、すべての指定されたラベルがノードに存在するかどうかを確認します (その値が何であるかを問わない)。

```
{"name" : "CheckNodeLabelPresence"}
```

checkServiceAffinity 述語は、ServiceAffinity ラベルがノードでスケジュールされる Pod について同種のものであることを確認します。

```
{"name" : "checkServiceAffinity"}
```

PodToleratesNodeNoExecuteTaints 述語は、Pod がノードの **NoExecute** テイントを容認できるかどうかを確認します。

```
{"name" : "PodToleratesNodeNoExecuteTaints"}
```

3.2.3.1.2. 汎用的な述語

以下の汎用的な述語は、非クリティカル述語とクリティカル述語が渡されるかどうかを確認します。非クリティカル述語は、非 Critical Pod のみが渡す必要のある述語であり、クリティカル述語はすべての Pod が渡す必要のある述語です。

デフォルトのスケジューラーポリシーにはこの汎用的な述語が含まれます。

汎用的な非クリティカル述語
PodFitsResources 述語は、リソースの可用性 (CPU、メモリー、GPU など) に基づいて適切な候補を判別します。ノードはそれらのリソース容量を宣言し、Pod は要求するリソースを指定できます。使用されるリソースではなく、要求されるリソースに基づいて適切な候補が判別されます。

```
{"name" : "PodFitsResources"}
```

汎用的なクリティカル述語
PodFitsHostPorts 述語は、ノードに要求される Pod ポートの空きポートがある (ポートの競合がない) かどうかを判別します。

```
{"name" : "PodFitsHostPorts"}
```

HostName 述語は、ホストパラメーターの有無と文字列のホスト名との一致に基づいて適切なノードを判別します。

```
{"name" : "HostName"}
```

MatchNodeSelector 述語は、Pod で定義されるノードセレクター (nodeSelector) のクエリーに基づいて適したノードを判別します。

```
{"name" : "MatchNodeSelector"}
```

3.2.3.2. スケジューラーの優先順位について

優先順位は、設定に応じてノードにランクを付けるルールです。

優先度のカスタムセットは、スケジューラーを設定するために指定できます。OpenShift Container Platform ではデフォルトでいくつかの優先度があります。他の優先度は、特定のパラメーターを指定してカスタマイズできます。優先順位に影響を与えるために、複数の優先度を組み合わせ、異なる重みをそれぞれのノードに指定することができます。

3.2.3.2.1. 静的優先度

静的優先度は、重みを除き、ユーザーからいずれの設定パラメーターも取りません。重みは指定する必要があり、0 または負の値にすることはできません。

これらは **openshift-config** プロジェクトのスケジューラーポリシー設定マップに指定されます。

3.2.3.2.1.1. デフォルトの優先度

デフォルトのスケジューラーポリシーには、以下の優先度が含まれています。それぞれの優先度関数は、重み **10000** を持つ **NodePreferAvoidPodsPriority** 以外は重み **1** を持ちます。

NodeAffinityPriority の優先度は、ノードアフィニティーのスケジュールの優先度に応じてノードに優先順位を付けます。

```
{"name" : "NodeAffinityPriority", "weight" : 1}
```

TaintTolerationPriority の優先度は、Pod についての **容認不可能な** テイント数の少ないノードを優先します。容認不可能なテイントとはキー **PreferNoSchedule** のあるテイントのことです。

```
{"name" : "TaintTolerationPriority", "weight" : 1}
```

ImageLocalityPriority の優先度は、Pod コンテナーのイメージをすでに要求しているノードを優先します。

```
{"name" : "ImageLocalityPriority", "weight" : 1}
```

SelectorSpreadPriority は、Pod に一致するサービス、レプリケーションコントローラー (RC)、レプリケーションセット (RS)、およびステートフルなセットを検索し、次にそれらのセレクターに一致する既存の Pod を検索します。スケジューラーは、一致する既存の Pod が少ないノードを優先します。次に、Pod のスケジュール時にそれらのセレクターに一致する Pod 数の最も少ないノードで Pod をスケジュールします。

```
{"name" : "SelectorSpreadPriority", "weight" : 1}
```

InterPodAffinityPriority の優先度は、ノードの対応する PodAffinityTerm が満たされている場合に **weightedPodAffinityTerm** の要素を使った繰り返し処理や **重み** の合計への追加によって合計を計算します。合計値の最も高いノードが最も優先されます。

```
{"name" : "InterPodAffinityPriority", "weight" : 1}
```

LeastRequestedPriority の優先度は、要求されたリソースの少ないノードを優先します。これは、ノードでスケジュールされる Pod によって要求されるメモリーおよび CPU のパーセンテージを計算し、利用可能な/残りの容量の値の最も高いノードを優先します。

```
{"name" : "LeastRequestedPriority", "weight" : 1}
```

BalancedResourceAllocation の優先度は、均衡が図られたリソース使用率に基づいてノードを優先します。これは、容量の一部として消費済み CPU とメモリー間の差異を計算し、2 つのメトリクスがどの程度相互に近似しているかに基づいてノードの優先度を決定します。これは常に **LeastRequestedPriority** と併用する必要があります。

```
{"name" : "BalancedResourceAllocation", "weight" : 1}
```

NodePreferAvoidPodsPriority の優先度は、レプリケーションコントローラー以外のコントローラーによって所有される Pod を無視します。

```
{"name" : "NodePreferAvoidPodsPriority", "weight" : 10000}
```

3.2.3.2.1.2. 他の静的優先度

OpenShift Container Platform は以下の優先度もサポートしています。

EqualPriority の優先度は、優先度の設定が指定されていない場合に、すべてのノードに等しい重み **1** を指定します。この優先順位はテスト環境にのみ使用することを推奨します。

```
{"name" : "EqualPriority", "weight" : 1}
```

MostRequestedPriority の優先度は、要求されたリソースの最も多いノードを優先します。これは、ノードスケジュールされる Pod で要求されるメモリーおよび CPU のパーセンテージを計算し、容量に対して要求される部分の平均の最大値に基づいて優先度を決定します。

```
{"name" : "MostRequestedPriority", "weight" : 1}
```

ServiceSpreadingPriority の優先度は、同じマシンに置かれる同じサービスに属する Pod 数を最小限にすることにより Pod を分散します。

```
{"name" : "ServiceSpreadingPriority", "weight" : 1}
```

3.2.3.2.2. 設定可能な優先順位

これらの優先順位を **openshift-config** namespace のスケジューラーポリシー設定マップに設定し、優先順位の機能に影響を与えるラベルを追加できます。

優先度関数のタイプは、それらが取る引数によって識別されます。これらは設定可能なため、ユーザー定義の名前が異なる場合に、同じタイプの (ただし設定パラメーターは異なる) 設定可能な複数の優先度を組み合わせることができます。

優先順位の使用方法については、スケジューラーポリシーの変更についての箇所を参照してください。

ServiceAntiAffinity の優先度はラベルを取り、ラベルの値に基づいてノードのグループ全体に同じサービスに属する Pod を適正に分散します。これは、指定されたラベルの同じ値を持つすべてのノードに同じスコアを付与します。また Pod が最も集中していないグループ内のノードにより高いスコアを付与します。

```
{
"kind": "Policy",
"apiVersion": "v1",

"priorities":[
   {
      "name":"<name>", (1)
      "weight" : 1 (2)
      "argument":{
         "serviceAntiAffinity":{
            "label": "<label>" (3)
            }
```

```
        }
      }
  ]
}
```

1. 優先度の名前を指定します。
2. 重みを指定します。ゼロ以外の正の値を指定します。
3. 一致するラベルを指定します。

以下に例を示します。

```
{
"kind": "Policy",
"apiVersion": "v1",
"priorities": [
   {
      "name":"RackSpread",
      "weight" : 1,
      "argument": {
         "serviceAntiAffinity": {
            "label": "rack"
            }
      }
   }
  ]
}
```

注記

カスタムラベルに基づいて **ServiceAntiAffinity** パラメーターを使用しても Pod を予想通りに展開できない場合があります。Red Hat ソリューション を参照してください。

labelPreference パラメーターは指定されたラベルに基づいて優先順位を指定します。ラベルがノードにある場合、そのノードに優先度が指定されます。ラベルが指定されていない場合は、優先度はラベルを持たないノードに指定されます。**labelPreference** パラメーターのある複数の優先度が設定されている場合、すべての優先度に同じ重みが付けられている必要があります。

```
{
"kind": "Policy",
"apiVersion": "v1",
"priorities":[
   {
      "name":"<name>", 1
      "weight" : 1 2
      "argument":{
         "labelPreference":{
            "label": "<label>", 3
            "presence": true 4
            }
         }
```

```
      }
    ]
}
```

1. 優先度の名前を指定します。
2. 重みを指定します。ゼロ以外の正の値を指定します。
3. 一致するラベルを指定します。
4. ラベルが必要であるかを、**true** または **false** のいずれかで指定します。

3.2.4. ポリシー設定のサンプル

以下の設定は、スケジューラーポリシーファイルを使って指定される場合のデフォルトのスケジューラー設定を示しています。

```
{
"kind": "Policy",
"apiVersion": "v1",
"predicates": [
    {
        "name": "RegionZoneAffinity", 1
        "argument": {
            "serviceAffinity": { 2
              "labels": ["region, zone"] 3
            }
        }
    }
  ],
"priorities": [
    {
        "name":"RackSpread", 4
        "weight" : 1,
        "argument": {
            "serviceAntiAffinity": { 5
                "label": "rack" 6
                }
            }
        }
    ]
}
```

1. 述語の名前です。
2. 述語のタイプです。
3. 述語のラベルです。
4. 優先順位の名前です。
5. 優先順位のタイプです。
6. 優先順位のラベルです。

以下の設定例のいずれの場合も、述語と優先度関数の一覧は、指定された使用例に関連するもののみを含むように切り捨てられます。実際には、完全な/分かりやすいスケジューラーポリシーには、上記のデフォルトの述語および優先度のほとんど (すべてではなくても) が含まれるはずです。

以下の例は、region (affinity) → zone (affinity) → rack (anti-affinity) の 3 つのトポロジーレベルを定義します。

```
{
"kind": "Policy",
"apiVersion": "v1",
"predicates": [
    {
        "name": "RegionZoneAffinity",
        "argument": {
            "serviceAffinity": {
              "labels": ["region, zone"]
            }
        }
    }
  ],
"priorities": [
    {
        "name":"RackSpread",
        "weight" : 1,
        "argument": {
            "serviceAntiAffinity": {
                "label": "rack"
                }
            }
        }
    ]
}
```

以下の例は、**city** (affinity) → **building** (anti-affinity) → **room** (anti-affinity) の 3 つのとポロジーレベルを定義します。

```
{
"kind": "Policy",
"apiVersion": "v1",
"predicates": [
    {
        "name": "CityAffinity",
        "argument": {
            "serviceAffinity": {
              "label": "city"
            }
        }
    }
  ],
"priorities": [
    {
        "name":"BuildingSpread",
        "weight" : 1,
        "argument": {
            "serviceAntiAffinity": {
```

```
                "label": "building"
                }
          }
       },
    {
        "name":"RoomSpread",
        "weight" : 1,
        "argument": {
            "serviceAntiAffinity": {
                "label": "room"
                }
          }
       }
   ]
}
```

以下の例では、region ラベルが定義されたノードのみを使用し、zone ラベルが定義されたノードを優先するポリシーを定義します。

```
{
"kind": "Policy",
"apiVersion": "v1",
"predicates": [
    {
        "name": "RequireRegion",
        "argument": {
            "labelPreference": {
                "labels": ["region"],
                "presence": true
          }
        }
     }
  ],
"priorities": [
    {
        "name":"ZonePreferred",
        "weight" : 1,
        "argument": {
            "labelPreference": {
                "label": "zone",
                "presence": true
                }
          }
       }
   ]
}
```

以下の例では、静的および設定可能な述語および優先順位を組み合わせています。

```
{
"kind": "Policy",
"apiVersion": "v1",
"predicates": [
    {
        "name": "RegionAffinity",
```

```
        "argument": {
          "serviceAffinity": {
            "labels": ["region"]
          }
        }
      },
      {
        "name": "RequireRegion",
        "argument": {
          "labelsPresence": {
            "labels": ["region"],
            "presence": true
          }
        }
      },
      {
        "name": "BuildingNodesAvoid",
        "argument": {
          "labelsPresence": {
            "label": "building",
            "presence": false
          }
        }
      },
      {"name" : "PodFitsPorts"},
      {"name" : "MatchNodeSelector"}
      ],
  "priorities": [
      {
        "name": "ZoneSpread",
        "weight" : 2,
        "argument": {
          "serviceAntiAffinity":{
            "label": "zone"
            }
          }
        },
      {
        "name":"ZonePreferred",
        "weight" : 1,
        "argument": {
          "labelPreference":{
            "label": "zone",
            "presence": true
            }
          }
        },
      {"name" : "ServiceSpreadingPriority", "weight" : 1}
      ]
  }
```

3.3. スケジューラープロファイルを使用した POD のスケジューリング

OpenShift Container Platform は、スケジューリングプロファイルを使用して Pod をクラスター内のノードにスケジュールするように設定できます。

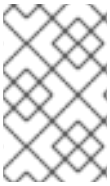
重要

スケジューラープロファイルは、テクノロジープレビュー機能としてのみご利用いただけます。テクノロジープレビュー機能は Red Hat の実稼働環境でのサービスレベルアグリーメント (SLA) ではサポートされていないため、Red Hat では実稼働環境での使用を推奨していません。Red Hat は実稼働環境でこれらを使用することを推奨していません。テクノロジープレビューの機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行いフィードバックを提供していただくことを目的としています。

Red Hat のテクノロジープレビュー機能のサポート範囲についての詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

3.3.1. スケジューラープロファイルについて

スケジューラープロファイルを指定して、Pod をノードにスケジュールする方法を制御できます。



注記

スケジューラープロファイルは、スケジューラーポリシーを設定する代わりに使用できます。スケジューラーポリシーとスケジューラープロファイルの両方は設定しないでください。両方が設定されている場合、スケジューラーポリシーが優先されます。

以下のスケジューラープロファイルを利用できます。

LowNodeUtilization

このプロファイルは、ノードごとのリソースの使用量を減らすためにノード間で Pod を均等に分散しようとします。このプロファイルは、デフォルトのスケジューラー動作を提供します。

HighNodeUtilization

このプロファイルは、できるだけ少ないノードにできるだけ多くの Pod を配置することを試行します。これによりノード数が最小限に抑えられ、ノードごとのリソースの使用率が高くなります。

NoScoring

これは、すべての Score プラグインを無効にして最速のスケジューリングサイクルを目指す低レイテンシープロファイルです。これにより、スケジューリングの高速化がスケジューリングにおける意思決定の質に対して優先されます。

3.3.2. スケジューラープロファイルの設定

スケジューラーがスケジューラープロファイルを使用するように設定できます。

注記

スケジューラーポリシーとスケジューラープロファイルの両方は設定しないでください。両方が設定されている場合、スケジューラーポリシーが優先されます。

前提条件

- **cluster-admin** ロールを持つユーザーとしてのクラスターへのアクセスがあること。

手順

1. **Scheduler** オブジェクトを編集します。

```
$ oc edit scheduler cluster
```

2. **spec.profile** フィールドで使用するプロファイルを指定します。

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  ...
  name: cluster
  resourceVersion: "601"
  selfLink: /apis/config.openshift.io/v1/schedulers/cluster
  uid: b351d6d0-d06f-4a99-a26b-87af62e79f59
spec:
  mastersSchedulable: false
  policy:
    name: ""
  profile: HighNodeUtilization (1)
```

(1) **LowNodeUtilization**、 **HighNodeUtilization**、または **NoScoring** に設定されます。

3. 変更を適用するためにファイルを保存します。

3.4. アフィニティールールと非アフィニティールールの使用による他の POD との相対での POD の配置

アフィニティーとは、スケジュールするノードを制御する Pod の特性です。非アフィニティーとは、Pod がスケジュールされることを拒否する Pod の特性です。

OpenShift Container Platform では、**Pod のアフィニティー** と **Pod の非アフィニティー** によって、他の Pod のキー/値ラベルに基づいて、Pod のスケジュールに適したノードを制限することができます。

3.4.1. Pod のアフィニティーについて

Pod のアフィニティー と **Pod の非アフィニティー** によって、他の Pod のキー/値ラベルに基づいて、Pod をスケジュールすることに適したノードを制限することができます。

- Pod のアフィニティーはスケジューラーに対し、新規 Pod のラベルセレクターが現在の Pod のラベルに一致する場合に他の Pod と同じノードで新規 Pod を見つけるように指示します。
- Pod の非アフィニティーは、新規 Pod のラベルセレクターが現在の Pod のラベルに一致する場合に、同じラベルを持つ Pod と同じノードで新規 Pod を見つけることを禁止します。

たとえば、アフィニティールールを使用することで、サービス内で、または他のサービスの Pod との関連で Pod を分散したり、パックしたりすることができます。非アフィニティールールにより、特定のサービスの Pod がそののサービスの Pod のパフォーマンスに干渉すると見なされる別のサービスの Pod と同じノードでスケジュールされることを防ぐことができます。または、関連する障害を減らすために複数のノードまたはアベイラビリティーゾーン間でサービスの Pod を分散することもできます。

Pod のアフィニティーには、**required (必須)** および **preferred (優先)** の 2 つのタイプがあります。

Pod をノードにスケジュールする前に、required (必須) ルールを **満たしている必要があります**。preferred (優先) ルールは、ルールを満たす場合に、スケジューラーはルールの実施を試行しますが、その実施が必ずしも保証される訳ではありません。

注記

Pod の優先順位およびプリエンプションの設定により、スケジューラーはアフィニティーの要件に違反しなければ Pod の適切なノードを見つけられない可能性があります。その場合、Pod はスケジュールされない可能性があります。

この状態を防ぐには、優先順位が等しい Pod との Pod のアフィニティーの設定を慎重に行ってください。

Pod のアフィニティー/非アフィニティーは **Pod** 仕様ファイルで設定します。required (必須) ルール、preferred (優先) ルールのいずれか、またはその両方を指定することができます。両方を指定する場合、ノードは最初に required (必須) ルールを満たす必要があり、その後に preferred (優先) ルールを満たそうとします。

以下の例は、Pod のアフィニティーおよび非アフィニティーに設定される **Pod** 仕様を示しています。

この例では、Pod のアフィニティールールは ノードにキー **security** と値 **S1** を持つラベルの付いた 1 つ以上の Pod がすでに実行されている場合にのみ Pod をノードにスケジュールできることを示しています。Pod の非アフィニティールールは、ノードがキー **security** と値 **S2** を持つラベルが付いた Pod がすでに実行されている場合は Pod をノードにスケジュールしないように設定することを示しています。

Pod のアフィニティーが設定された Pod 設定ファイルのサンプル

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity: 1
      requiredDuringSchedulingIgnoredDuringExecution: 2
      - labelSelector:
          matchExpressions:
          - key: security 3
            operator: In 4
            values:
            - S1 5
        topologyKey: failure-domain.beta.kubernetes.io/zone
  containers:
  - name: with-pod-affinity
    image: docker.io/ocpqe/hello-pod
```

1. Pod のアフィニティーを設定するためのスタンザです。
2. required (必須) ルールを定義します。
3. 5. ルールを適用するために一致している必要のあるキーと値 (ラベル) です。
4. 演算子は、既存 Pod のラベルと新規 Pod の仕様の **matchExpression** パラメーターの値のセットの間の関係を表します。これには **In**、**NotIn**、**Exists**、または **DoesNotExist** のいずれかを使用できます。

Pod の非アフィニティーが設定された Pod 設定ファイルのサンプル

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-antiaffinity
spec:
  affinity:
    podAntiAffinity: (1)
      preferredDuringSchedulingIgnoredDuringExecution: (2)
      - weight: 100 (3)
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: security (4)
              operator: In (5)
              values:
              - S2
          topologyKey: kubernetes.io/hostname
  containers:
  - name: with-pod-affinity
    image: docker.io/ocpqe/hello-pod
```

(1) Pod の非アフィニティーを設定するためのスタンザです。

(2) preferred (優先) ルールを定義します。

(3) preferred (優先) ルールの重みを指定します。最も高い重みを持つノードが優先されます。

(4) 非アフィニティールールが適用される時を決定する Pod ラベルの説明です。ラベルのキーおよび値を指定します。

(5) 演算子は、既存 Pod のラベルと新規 Pod の仕様の **matchExpression** パラメーターの値のセットの間の関係を表します。これには **In**、**NotIn**、**Exists**、または **DoesNotExist** のいずれかを使用できます。

注記

ノードのラベルに、Pod のノードのアフィニティールールを満たさなくなるような結果になる変更がランタイム時に生じる場合も、Pod はノードで引き続き実行されます。

3.4.2. Pod アフィニティールールの設定

以下の手順は、ラベルの付いた Pod と Pod のスケジュールを可能にするアフィニティーを使用する Pod を作成する 2 つの Pod の単純な設定を示しています。

手順

1. **Pod** 仕様の特定のラベルの付いた Pod を作成します。

```
$ cat team4.yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-s1
```

```
  labels:
    security: S1
spec:
  containers:
  - name: security-s1
    image: docker.io/ocpqe/hello-pod
```

2. 他の Pod の作成時に、以下のように **Pod** 仕様を編集します。

   a. **podAffinity** スタンザを使用して、**requiredDuringSchedulingIgnoredDuringExecution** パラメーターまたは **preferredDuringSchedulingIgnoredDuringExecution** パラメーターを設定します。

   b. 満たしている必要のあるキーおよび値を指定します。新規 Pod を他の Pod と共にスケジュールする必要がある場合、最初の Pod のラベルと同じ **key** および **value** パラメーターを使用します。

```
podAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
  - labelSelector:
      matchExpressions:
      - key: security
        operator: In
        values:
        - S1
    topologyKey: failure-domain.beta.kubernetes.io/zone
```

   c. **operator** を指定します。演算子は **In**、**NotIn**、**Exists**、または **DoesNotExist** にすることができます。たとえば、演算子 **In** を使用してラベルをノードで必要になるようにします。

   d. **topologyKey** を指定します。これは、システムがトポロジードメインを表すために使用する事前にデータが設定された Kubernetes ラベル です。

3. Pod を作成します。

```
$ oc create -f <pod-spec>.yaml
```

3.4.3. Pod 非アフィニティールールの設定

以下の手順は、ラベルの付いた Pod と Pod のスケジュールの禁止を試行する非アフィニティーの preferred (優先) ルールを使用する Pod を作成する 2 つの Pod の単純な設定を示しています。

手順

1. **Pod** 仕様の特定のラベルの付いた Pod を作成します。

```
$ cat team4.yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-s2
  labels:
    security: S2
spec:
```

```
containers:
- name: security-s2
  image: docker.io/ocpqe/hello-pod
```

2. 他の Pod の作成時に、**Pod** 仕様を編集して以下のパラメーターを設定します。

3. **podAntiAffinity** スタンザを使用して、**requiredDuringSchedulingIgnoredDuringExecution** パラメーターまたは **preferredDuringSchedulingIgnoredDuringExecution** パラメーターを設定します。

   a. ノードの重みを 1-100 で指定します。最も高い重みを持つノードが優先されます。

   b. 満たしている必要のあるキーおよび値を指定します。新規 Pod を他の Pod と共にスケジュールされないようにする必要がある場合、最初の Pod のラベルと同じ **key** および **value** パラメーターを使用します。

```
podAntiAffinity:
  preferredDuringSchedulingIgnoredDuringExecution:
  - weight: 100
    podAffinityTerm:
      labelSelector:
        matchExpressions:
        - key: security
          operator: In
          values:
          - S2
      topologyKey: kubernetes.io/hostname
```

   c. preferred (優先) ルールの場合、重みを 1-100 で指定します。

   d. **operator** を指定します。演算子は **In**、**NotIn**、**Exists**、または **DoesNotExist** にすることができます。たとえば、演算子 **In** を使用してラベルをノードで必要になるようにします。

4. **topologyKey** を指定します。これは、システムがトポロジードメインを表すために使用する事前にデータが設定された Kubernetes ラベル です。

5. Pod を作成します。

```
$ oc create -f <pod-spec>.yaml
```

3.4.4. Pod のアフィニティールールと非アフィニティールールの例

以下の例は、Pod のアフィニティーおよび非アフィニティーについて示しています。

3.4.4.1. Pod のアフィニティー

以下の例は、一致するラベルとラベルセレクターを持つ Pod についての Pod のアフィニティーを示しています。

- Pod **team4** にはラベル **team:4** が付けられています。

```
$ cat team4.yaml
apiVersion: v1
kind: Pod
metadata:
```

```
  name: team4
  labels:
    team: "4"
spec:
  containers:
  - name: ocp
    image: docker.io/ocpqe/hello-pod
```

- Pod **team4a** には、**podAffinity** の下にラベルセレクター **team:4** が付けられています。

```
$ cat pod-team4a.yaml
apiVersion: v1
kind: Pod
metadata:
  name: team4a
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: team
            operator: In
            values:
            - "4"
        topologyKey: kubernetes.io/hostname
  containers:
  - name: pod-affinity
    image: docker.io/ocpqe/hello-pod
```

- **team4a** Pod は **team4** Pod と同じノードにスケジュールされます。

3.4.4.2. Pod の非アフィニティー

以下の例は、一致するラベルとラベルセレクターを持つ Pod についての Pod の非アフィニティーを示しています。

- Pod **pod-s1** にはラベル **security:s1** が付けられています。

```
cat pod-s1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-s1
  labels:
    security: s1
spec:
  containers:
  - name: ocp
    image: docker.io/ocpqe/hello-pod
```

- Pod **pod-s2** には、**podAntiAffinity** の下にラベルセレクター **security:s1** が付けられています。

```
cat pod-s2.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-s2
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - s1
        topologyKey: kubernetes.io/hostname
  containers:
  - name: pod-antiaffinity
    image: docker.io/ocpqe/hello-pod
```

- Pod **pod-s2** は **pod-s1** と同じノードにスケジュールできません。

3.4.4.3. 一致するラベルのない Pod のアフィニティー

以下の例は、一致するラベルとラベルセレクターのない Pod についての Pod のアフィニティーを示しています。

- Pod **pod-s1** にはラベル **security:s1** が付けられています。

```
$ cat pod-s1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-s1
  labels:
    security: s1
spec:
  containers:
  - name: ocp
    image: docker.io/ocpqe/hello-pod
```

- Pod **pod-s2** にはラベルセレクター **security:s2** があります。

```
$ cat pod-s2.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-s2
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
```

```
          values:
          - s2
      topologyKey: kubernetes.io/hostname
containers:
- name: pod-affinity
  image: docker.io/ocpqe/hello-pod
```

- Pod **pod-s2** は、**security:s2** ラベルの付いた Pod を持つノードがない場合はスケジュールされません。そのラベルの付いた他の Pod がない場合、新規 Pod は保留状態のままになります。

出力例

```
NAME     READY    STATUS    RESTARTS   AGE      IP       NODE
pod-s2   0/1      Pending   0          32s      <none>
```

3.5. ノードのアフィニティールールを使用したノード上での POD 配置の制御

アフィニティーとは、スケジュールするノードを制御する Pod の特性です。

OpenShift Container Platformnode では、アフィニティーとはスケジューラーが Pod を配置する場所を決定するために使用する一連のルールのことです。このルールは、ノードのカスタムラベルと Pod で指定されたラベルセレクターを使って定義されます。

3.5.1. ノードのアフィニティーについて

ノードのアフィニティーにより、Pod がその配置に使用できるノードのグループに対してアフィニティーを指定できます。ノード自体は配置に対して制御を行いません。

たとえば、Pod を特定の CPU を搭載したノードまたは特定のアベイラビリティーゾーンにあるノードでのみ実行されるよう設定することができます。

ノードのアフィニティールールには、**required (必須)** および **preferred (優先)** の 2 つのタイプがあります。

Pod をノードにスケジュールする前に、required (必須) ルールを **満たしている必要があります**。preferred (優先) ルールは、ルールを満たす場合に、スケジューラーはルールの実施を試行しますが、その実施が必ずしも保証される訳ではありません。

注記

ランタイム時にノードのラベルに変更が生じ、その変更により Pod でのノードのアフィニティールールを満たさなくなる状態が生じるでも、Pod はノードで引き続き実行されます。

ノードのアフィニティーは **Pod** 仕様ファイルで設定します。required (必須) ルール、preferred (優先) ルールのいずれか、またはその両方を指定することができます。両方を指定する場合、ノードは最初に required (必須) ルールを満たす必要があり、その後に preferred (優先) ルールを満たそうとします。

以下の例は、Pod をキーが **e2e-az-NorthSouth** で、その値が **e2e-az-North** または **e2e-az-South** のいずれかであるラベルの付いたノードに Pod を配置することを求めるルールが設定された **Pod** 仕様です。

ノードのアフィニティーの required (必須) ルールが設定された Pod 設定ファイルのサンプル

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity: 1
      requiredDuringSchedulingIgnoredDuringExecution: 2
        nodeSelectorTerms:
        - matchExpressions:
          - key: e2e-az-NorthSouth 3
            operator: In 4
            values:
            - e2e-az-North 5
            - e2e-az-South 6
  containers:
  - name: with-node-affinity
    image: docker.io/ocpqe/hello-pod
```

1 ノードのアフィニティーを設定するためのスタンザです。

2 required (必須) ルールを定義します。

3 5 6 ルールを適用するために一致している必要のあるキー/値のペア (ラベル) です。

4 演算子は、ノードのラベルと **Pod** 仕様の **matchExpression** パラメーターの値のセットの間の関係を表します。この値は、**In**、**NotIn**、**Exists**、または **DoesNotExist**、**Lt**、または **Gt** にすることができます。

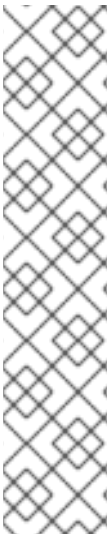
以下の例は、キーが **e2e-az-EastWest** で、その値が **e2e-az-East** または **e2e-az-West** のラベルが付いたノードに Pod を配置すること優先する preferred (優先) ルールが設定されたノード仕様です。

ノードのアフィニティーの preferred (優先) ルールが設定された Pod 設定ファイルのサンプル

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity: 1
      preferredDuringSchedulingIgnoredDuringExecution: 2
      - weight: 1 3
        preference:
          matchExpressions:
          - key: e2e-az-EastWest 4
            operator: In 5
            values:
            - e2e-az-East 6
            - e2e-az-West 7
  containers:
  - name: with-node-affinity
    image: docker.io/ocpqe/hello-pod
```

1. ノードのアフィニティーを設定するためのスタンザです。
2. preferred (優先) ルールを定義します。
3. preferred (優先) ルールの重みを指定します。最も高い重みを持つノードが優先されます。
4. 6. 7. ルールを適用するために一致している必要のあるキー/値のペア (ラベル) です。
5. 演算子は、ノードのラベルと **Pod** 仕様の **matchExpression** パラメーターの値のセットの間の関係を表します。この値は、**In**、**NotIn**、**Exists**、または **DoesNotExist**、**Lt**、または **Gt** にすることができます。

ノードの非アフィニティー についての明示的な概念はありませんが、**NotIn** または **DoesNotExist** 演算子を使用すると、動作が複製されます。



注記

同じ Pod 設定でノードのアフィニティーとノードのセレクターを使用している場合は、以下に注意してください。

- **nodeSelector** と **nodeAffinity** の両方を設定する場合、Pod が候補ノードでスケジュールされるにはどちらの条件も満たしている必要があります。
- **nodeAffinity** タイプに関連付けられた複数の **nodeSelectorTerms** を指定する場合、**nodeSelectorTerms** のいずれかが満たされている場合に Pod をノードにスケジュールすることができます。
- **nodeSelectorTerms** に関連付けられた複数の **matchExpressions** を指定する場合、すべての **matchExpressions** が満たされている場合にのみ Pod をノードにスケジュールすることができます。

3.5.2. ノードアフィニティーの required (必須) ルールの設定

Pod をノードにスケジュールする前に、required (必須) ルールを **満たしている必要があります**。

手順

以下の手順は、ノードとスケジューラーがノードに配置する必要のある Pod を作成する単純な設定を示しています。

1. **oc label node** コマンドを使ってラベルをノードに追加します。

```
$ oc label node node1 e2e-az-name=e2e-az1
```

2. **Pod** 仕様では、**nodeAffinity** スタンザを使用して **requiredDuringSchedulingIgnoredDuringExecution** パラメーターを設定します。
   a. 満たしている必要のあるキーおよび値を指定します。新規 Pod を編集したノードにスケジュールする必要がある場合、ノードのラベルと同じ **key** および **value** パラメーターを使用します。
   b. **operator** を指定します。演算子は **In**、**NotIn**、**Exists**、**DoesNotExist**、**Lt**、または **Gt** にすることができます。たとえば、演算子 **In** を使用してラベルがノードで必要になるようにします。

出力例

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
```

3. Pod を作成します。

```
$ oc create -f e2e-az2.yaml
```

3.5.3. ノードアフィニティーの preferred (優先) ルールの設定

preferred (優先) ルールは、ルールを満たす場合に、スケジューラーはルールの実施を試行しますが、その実施が必ずしも保証される訳ではありません。

手順

以下の手順は、ノードとスケジューラーがノードに配置しようとする Pod を作成する単純な設定を示しています。

1. **oc label node** コマンドを使ってラベルをノードに追加します。

```
$ oc label node node1 e2e-az-name=e2e-az3
```

2. **Pod** 仕様では、 **nodeAffinity** スタンザを使用して **preferredDuringSchedulingIgnoredDuringExecution** パラメーターを設定します。

   a. ノードの重みを数字の 1-100 で指定します。最も高い重みを持つノードが優先されます。

   b. 満たしている必要のあるキーおよび値を指定します。新規 Pod を編集したノードにスケジュールする必要がある場合、ノードのラベルと同じ **key** および **value** パラメーターを使用します。

```
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: e2e-az-name
            operator: In
            values:
            - e2e-az3
```

   c. **operator** を指定します。演算子は **In**、**NotIn**、**Exists**、**DoesNotExist**、**Lt**、または **Gt** にすることができます。たとえば、演算子 **In** を使用してラベルがノードで必要になるようにします。

3. Pod を作成します。

```
$ oc create -f e2e-az3.yaml
```

3.5.4. ノードのアフィニティールールの例

以下の例は、ノードのアフィニティーを示しています。

3.5.4.1. 一致するラベルを持つノードのアフィニティー

以下の例は、一致するラベルを持つノードと Pod のノードのアフィニティーを示しています。

- Node1 ノードにはラベル **zone:us** があります。

```
$ oc label node node1 zone=us
```

- pod-s1 pod にはノードアフィニティーの required (必須) ルールの下に **zone** と **us** のキー/値のペアがあります。

```
$ cat pod-s1.yaml
```

出力例

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-s1
spec:
  containers:
    - image: "docker.io/ocpqe/hello-pod"
      name: hello-pod
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
            - key: "zone"
              operator: In
              values:
              - us
```

- pod-s1 pod は Node1 でスケジュールできます。

```
$ oc get pod -o wide
```

出力例

```
NAME    READY    STATUS      RESTARTS   AGE     IP      NODE
pod-s1  1/1      Running     0          4m      IP1     node1
```

3.5.4.2. 一致するラベルのないノードのアフィニティー

以下の例は、一致するラベルを持たないノードと Pod のノードのアフィニティーを示しています。

- Node1 ノードにはラベル **zone:emea** があります。

```
$ oc label node node1 zone=emea
```

- pod-s1 pod にはノードアフィニティーの required (必須) ルールの下に **zone** と **us** のキー/値のペアがあります。

```
$ cat pod-s1.yaml
```

出力例

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-s1
spec:
  containers:
    - image: "docker.io/ocpqe/hello-pod"
      name: hello-pod
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
            - key: "zone"
              operator: In
              values:
              - us
```

- pod-s1 pod は Node1 でスケジュールすることができません。

```
$ oc describe pod pod-s1
```

出力例

```
...

Events:
 FirstSeen LastSeen Count From              SubObjectPath  Type               Reason
 --------- -------- ----- ----              -------------  --------           ------
 1m        33s      8     default-scheduler Warning        FailedScheduling   No nodes are
available that match all of the following predicates:: MatchNodeSelector (1).
```

3.5.5. 関連情報

- ノードラベルの変更に関する詳細は、ノードでラベルを更新する方法について を参照してください。

3.6. POD のオーバーコミットノードへの配置

オーバーコミット とは、コンテナーの計算リソース要求と制限の合計が、そのシステムで利用できるリソースを超えた状態のことです。オーバーコミットは、容量に対して保証されたパフォーマンスのトレードオフが許容可能である開発環境において、望ましいことがあります。

要求および制限により、管理者はノードでのリソースのオーバーコミットを許可し、管理できます。スケジューラーは、要求を使ってコンテナーをスケジュールし、最小限のサービス保証を提供します。制限は、ノード上で消費されるコンピュートリソースの量を制限します。

3.6.1. オーバーコミットについて

要求および制限により、管理者はノードでのリソースのオーバーコミットを許可し、管理できます。スケジューラーは、要求を使ってコンテナーをスケジュールし、最小限のサービス保証を提供します。制限は、ノード上で消費されるコンピュートリソースの量を制限します。

OpenShift Container Platform 管理者は、開発者がコンテナーで設定された要求と制限の比率を上書きするようマスターを設定することで、オーバーコミットのレベルを制御し、ノードのコンテナー密度を管理します。この設定を、制限とデフォルトを指定するプロジェクトごとの **LimitRange** と共に使用することで、オーバーコミットを必要なレベルに設定できるようコンテナーの制限と要求を調整することができます。

注記

コンテナーに制限が設定されていない場合には、これらの上書きは影響を与えません。デフォルトの制限で (個別プロジェクトごとに、またはプロジェクトテンプレートを使用して) **LimitRange** オブジェクトを作成し、上書きが適用されるようにします。

上書き後も、コンテナーの制限および要求は、プロジェクトのいずれかの **LimitRange** オブジェクトで引き続き検証される必要があります。たとえば、開発者が最小限度に近い制限を指定し、要求を最小限度よりも低い値に上書きすることで、Pod が禁止される可能性があります。この最適でないユーザーエクスペリエンスについては、今後の作業で対応する必要がありますが、現時点ではこの機能および **LimitRange** オブジェクトを注意して設定してください。

3.6.2. ノードのオーバーコミットについて

オーバーコミット環境では、最適なシステム動作を提供できるようにノードを適切に設定する必要があります。

ノードが起動すると、メモリー管理用のカーネルの調整可能なフラグが適切に設定されます。カーネルは、物理メモリーが不足しない限り、メモリーの割り当てに失敗するこはありません。

この動作を確認するため、OpenShift Container Platform は、**vm.overcommit_memory** パラメーターを **1** に設定し、デフォルトのオペレーティングシステムの設定を上書きすることで、常にメモリーをオーバーコミットするようにカーネルを設定します。

また、OpenShift Container Platform は **vm.panic_on_oom** パラメーターを **0** に設定することで、メモリーが不足したときでもカーネルがパニックにならないようにします。0 の設定は、Out of Memory (OOM) 状態のときに oom_killer を呼び出すようカーネルに指示します。これにより、優先順位に基づいてプロセスを強制終了します。

現在の設定は、ノードに以下のコマンドを実行して表示できます。

```
$ sysctl -a |grep commit
```

出力例

```
```

```
vm.overcommit_memory = 1
```

```
$ sysctl -a |grep panic
```

出力例

```
vm.panic_on_oom = 0
```

注記

上記のフラグはノード上にすでに設定されているはずであるため、追加のアクションは不要です。

各ノードに対して以下の設定を実行することもできます。

- CPU CFS クォータを使用した CPU 制限の無効化または実行
- システムプロセスのリソース予約
- Quality of Service (QoS) 層でのメモリー予約

3.7. ノードテイントを使用した POD 配置の制御

テイントおよび容認 (Toleration) により、ノードはノード上でスケジュールする必要のある (またはスケジュールすべきでない) Pod を制御できます。

3.7.1. テイントおよび容認 (Toleration) について

テイント により、ノードは Pod に一致する **容認** がない場合に Pod のスケジュールを拒否することができます。

テイントは **Node** 仕様 (**NodeSpec**) でノードに適用され、容認は **Pod** 仕様 (**PodSpec**) で Pod に適用されます。テイントをノードに適用する場合、スケジューラーは Pod がテイントを容認しない限り、Pod をそのノードに配置することができません。

ノード仕様のテイントの例

```
spec:
....
  template:
....
    spec:
      taints:
      - effect: NoExecute
        key: key1
        value: value1
....
```

Pod 仕様での容認の例

```
spec:
....
```

```
  template:
....
    spec:
      tolerations:
      - key: "key1"
        operator: "Equal"
        value: "value1"
        effect: "NoExecute"
        tolerationSeconds: 3600
....
```

テイントおよび容認は、key、value、および effect で設定されています。

表3.1テイントおよび容認コンポーネント

パラメーター	説明
key	**key** には、253 文字までの文字列を使用できます。キーは文字または数字で開始する必要があり、文字、数字、ハイフン、ドットおよびアンダースコアを含めることができます。
value	**value** には、63 文字までの文字列を使用できます。値は文字または数字で開始する必要があり、文字、数字、ハイフン、ドットおよびアンダースコアを含めることができます。
effect	effect は以下のいずれかにすることができます。 **NoSchedule** [1] • テイントに一致しない新規 Pod はノードにスケジュールされません。 • ノードの既存 Pod はそのままになります。 **PreferNoSchedule** • テイントに一致しない新規 Pod はノードにスケジュールされる可能性がありますが、スケジューラーはスケジュールしないようにします。 • ノードの既存 Pod はそのままになります。 **NoExecute** • テイントに一致しない新規 Pod はノードにスケジュールできません。 • 一致する容認を持たないノードの既存 Pod は削除されます。

パラメーター	説明	
operator	**Equal**	**key**/**value**/**effect** パラメーターは一致する必要があります。これはデフォルトになります。
	Exists	**key**/**effect** パラメーターは一致する必要があります。いずれかに一致する **value** パラメーターを空のままにする必要があります。

1. **NoSchedule** テイントをコントロールプレーンノード (別名マスターノード) に追加する場合、ノードには、デフォルトで追加される **node-role.kubernetes.io/master=:NoSchedule** テイントが必要です。
   以下に例を示します。

```
apiVersion: v1
kind: Node
metadata:
  annotations:
    machine.openshift.io/machine: openshift-machine-api/ci-ln-62s7gtb-f76d1-v8jxv-master-0
    machineconfiguration.openshift.io/currentConfig: rendered-master-
cdc1ab7da414629332cc4c3926e6e59c
...
spec:
  taints:
  - effect: NoSchedule
    key: node-role.kubernetes.io/master
...
```

容認はテイントと一致します。

- **operator** パラメーターが **Equal** に設定されている場合:
  - **key** パラメーターは同じになります。
  - **value** パラメーターは同じになります。
  - **effect** パラメーターは同じになります。
- **operator** パラメーターが **Exists** に設定されている場合:
  - **key** パラメーターは同じになります。
  - **effect** パラメーターは同じになります。

以下のテイントは OpenShift Container Platform に組み込まれています。

- **node.kubernetes.io/not-ready**: ノードは準備状態にありません。これはノード条件 **Ready=False** に対応します。

- **node.kubernetes.io/unreachable**: ノードはノードコントローラーから到達不能です。これはノード条件 **Ready=Unknown** に対応します。
- **node.kubernetes.io/memory-pressure**: ノードにはメモリー不足の問題が発生しています。これはノード条件 **MemoryPressure=True** に対応します。
- **node.kubernetes.io/disk-pressure**: ノードにはディスク不足の問題が発生しています。これはノード条件 **DiskPressure=True** に対応します。
- **node.kubernetes.io/network-unavailable**: ノードのネットワークは使用できません。
- **node.kubernetes.io/unschedulable**: ノードはスケジュールが行えません。
- **node.cloudprovider.kubernetes.io/uninitialized**: ノードコントローラーが外部のクラウドプロバイダーを使って起動すると、このテイントはノード上に設定され、使用不可能とマークされます。cloud-controller-manager のコントローラーがこのノードを初期化した後に、kubelet がこのテイントを削除します。
- **node.kubernetes.io/pid-pressure**: ノードが pid 不足の状態です。これはノード条件 **PIDPressure=True** に対応します。

重要

OpenShift Container Platform では、デフォルトの pid.available **evictionHard** は設定されません。

3.7.1.1. Pod のエビクションを遅延させる容認期間 (秒数) の使用方法

Pod 仕様または **MachineSet** に **tolerationSeconds** パラメーターを指定して、Pod がエビクションされる前にノードにバインドされる期間を指定できます。effect が **NoExecute** のテイントがノードに追加される場合、テイントを容認する Pod に **tolerationSeconds** パラメーターがある場合、Pod は期限切れになるまでエビクトされません。

出力例

```
spec:
....
  template:
....
    spec:
      tolerations:
      - key: "key1"
        operator: "Equal"
        value: "value1"
        effect: "NoExecute"
        tolerationSeconds: 3600
```

ここで、この Pod が実行中であるものの、一致する容認がない場合、Pod は 3,600 秒間バインドされたままとなり、その後にエビクトされます。テイントが期限前に削除される場合、Pod はエビクトされません。

3.7.1.2. 複数のテイントの使用方法

複数のテイントを同じノードに、複数の容認を同じ Pod に配置することができます。OpenShift Container Platform は複数のテイントと容認を以下のように処理します。

1. Pod に一致する容認のあるテイントを処理します。
2. 残りの一致しないテイントは Pod について以下の effect を持ちます。
   - effect が **NoSchedule** の一致しないテイントが 1 つ以上ある場合、OpenShift Container Platform は Pod をノードにスケジュールできません。
   - effect が **NoSchedule** の一致しないテイントがなく、effect が **PreferNoSchedule** の一致しない テイントが 1 つ以上ある場合、OpenShift Container Platform は Pod のノードへのスケジュールを試行しません。
   - effect が **NoExecute** のテイントが 1 つ以上ある場合、OpenShift Container Platform は Pod をノードからエビクトするか (ノードですでに実行中の場合)、または Pod のそのノードへのスケジュールが実行されません (ノードでまだ実行されていない場合)。
     - テイントを容認しない Pod はすぐにエビクトされます。
     - **Pod** の仕様に **tolerationSeconds** を指定せずにテイントを容認する Pod は永久にバインドされたままになります。
     - 指定された **tolerationSeconds** を持つテイントを容認する Pod は指定された期間バインドされます。

以下に例を示します。

- 以下のテイントをノードに追加します。

```
$ oc adm taint nodes node1 key1=value1:NoSchedule
```

```
$ oc adm taint nodes node1 key1=value1:NoExecute
```

```
$ oc adm taint nodes node1 key2=value2:NoSchedule
```

- Pod には以下の容認があります。

```
spec:
....
  template:
....
    spec:
      tolerations:
      - key: "key1"
        operator: "Equal"
        value: "value1"
        effect: "NoSchedule"
      - key: "key1"
        operator: "Equal"
        value: "value1"
        effect: "NoExecute"
```

この場合、3 つ目のテイントに一致する容認がないため、Pod はノードにスケジュールできません。Pod はこのテイントの追加時にノードですでに実行されている場合は実行が継続されます。3 つ目のテイントは 3 つのテイントの中で Pod で容認されない唯一のテイントであるためです。

3.7.1.3. Pod のスケジューリングとノードの状態 (Taint Nodes By Condition) について

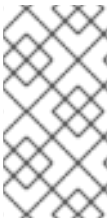
Taint Nodes By Condition (状態別のノードへのテイント) 機能はデフォルトで有効にされており、これはメモリー不足やディスク不足などの状態を報告するノードを自動的にテイントします。ノードが状態を報告すると、その状態が解消するまでテイントが追加されます。テイントに **NoSchedule** の effect がある場合、ノードが一致する容認を持つまでそのノードに Pod をスケジュールすることはできません。

スケジューラーは、Pod をスケジュールする前に、ノードでこれらのテイントの有無をチェックします。テイントがある場合、Pod は別のノードにスケジュールされます。スケジューラーは実際のノードの状態ではなくテイントをチェックするので、適切な Pod 容認を追加して、スケジューラーがこのようなノードの状態を無視するように設定します。

デーモンセットコントローラーは、以下の容認をすべてのデーモンに自動的に追加し、下位互換性を確保します。

- node.kubernetes.io/memory-pressure
- node.kubernetes.io/disk-pressure
- node.kubernetes.io/unschedulable (1.10 以降)
- node.kubernetes.io/network-unavailable (ホストネットワークのみ)

デーモンセットには任意の容認を追加することも可能です。



注記

コントロールプレーンは、QoS クラスを持つ Pod に **node.kubernetes.io/memory-pressure** 容認も追加します。これは、Kubernetes が **Guaranteed** または **Burstable** QoS クラスで Pod を管理するためです。新しい **BestEffort** Pod は、影響を受けるノードにスケジュールされません。

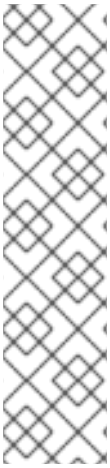
3.7.1.4. Pod の状態別エビクションについて (Taint-Based Eviction)

Taint-Based Eviction 機能はデフォルトで有効にされており、これは **not-ready** および **unreachable** などの特定の状態にあるノードから Pod をエビクトします。ノードがこうした状態のいずれかになると、OpenShift Container Platform はテイントをノードに自動的に追加して、Pod のエビクトおよび別のノードでの再スケジュールを開始します。

Taint Based Eviction には **NoExecute** の effect があり、そのテイントを容認しない Pod はすぐにエビクトされ、これを容認する Pod はエビクトされません (Pod が **tolerationSeconds** パラメーターを使用しない場合に限ります)。

tolerationSeconds パラメーターを使用すると、ノード状態が設定されたノードに Pod がどの程度の期間バインドされるかを指定することができます。**tolerationSeconds** の期間後もこの状態が続くと、テイントはノードに残り続け、一致する容認を持つ Pod はエビクトされます。**tolerationSeconds** の期間前にこの状態が解消される場合、一致する容認を持つ Pod は削除されません。

値なしで **tolerationSeconds** パラメーターを使用する場合、Pod は not ready(準備未完了) および unreachable(到達不能) のノードの状態が原因となりエビクトされることはありません。



注記

OpenShift Container Platform は、レートが制限された方法で Pod をエビクトし、マスターがノードからパーティション化される場合などのシナリオで発生する大規模な Pod エビクションを防ぎます。

デフォルトでは、特定のゾーン内のノードの 55% 以上が **異常である場合、ノードライフサイクルコントローラーはそのゾーンの状態を PartialDisruption** に変更し、Pod の削除率が低下します。この状態の小さなクラスター (デフォルトでは 50 ノード以下) の場合、このゾーンのノードは汚染されず、排除が停止されます。

詳細については、Kubernetes ドキュメントの Rate limits on eviction を参照してください。

OpenShift Container Platform は、**node.kubernetes.io/not-ready** および **node.kubernetes.io/unreachable** の容認を、**Pod** 設定がいずれかの容認を指定しない限り、自動的に **tolerationSeconds=300** に追加します。

```
spec:
....
  template:
....
    spec:
      tolerations:
      - key: node.kubernetes.io/not-ready
        operator: Exists
        effect: NoExecute
        tolerationSeconds: 300 (1)
      - key: node.kubernetes.io/unreachable
        operator: Exists
        effect: NoExecute
        tolerationSeconds: 300
```

(1) これらの容認は、ノード状態の問題のいずれかが検出された後、デフォルトの Pod 動作のバインドを 5 分間維持できるようにします。

これらの容認は必要に応じて設定できます。たとえば、アプリケーションに多数のローカル状態がある場合、ネットワークのパーティション化などに伴い、Pod をより長い時間ノードにバインドさせる必要があるかもしれません。 これにより、パーティションを回復させることができ、Pod のエビクションを回避できます。

デーモンセットによって起動する Pod は、**tolerationSeconds** が指定されない以下のテイントの **NoExecute** 容認を使用して作成されます。

- **node.kubernetes.io/unreachable**
- **node.kubernetes.io/not-ready**

その結果、デーモンセット Pod は、これらのノードの状態が原因でエビクトされることはありません。

3.7.1.5. すべてのテイントの許容

ノードは、**operator: "Exists"** 容認を **key** および **value** パラメーターなしで追加することですべてのテイントを容認するように Pod を設定できます。この容認のある Pod はテイントを持つノードから削除されません。

すべてのテイントを容認するための Pod 仕様

```
spec:
....
  template:
....
    spec:
      tolerations:
      - operator: "Exists"
```

3.7.2. テイントおよび容認 (Toleration) の追加

容認を Pod に、テイントをノードに追加することで、ノードはノード上でスケジュールする必要のある (またはスケジュールすべきでない) Pod を制御できます。既存の Pod およびノードの場合、最初に容認を Pod に追加してからテイントをノードに追加して、容認を追加する前に Pod がノードから削除されないようにする必要があります。

手順

1. **Pod** 仕様を **tolerations** スタンザを含めるように編集して、容認を Pod に追加します。

   Equal 演算子を含む Pod 設定ファイルのサンプル

   ```
   spec:
   ....
     template:
   ....
       spec:
         tolerations:
         - key: "key1" (1)
           value: "value1"
           operator: "Equal"
           effect: "NoExecute"
           tolerationSeconds: 3600 (2)
   ```

   (1) **テイントおよび容認コンポーネント** の表で説明されている toleration パラメーターです。

   (2) **tolerationSeconds** パラメーターは、エビクトする前に Pod をどの程度の期間ノードにバインドさせるかを指定します。

   以下に例を示します。

   Exists 演算子を含む Pod 設定ファイルのサンプル

   ```
   spec:
   ....
     template:
   ....
       spec:
   ```

```
tolerations:
- key: "key1"
  operator: "Exists" (1)
  effect: "NoExecute"
  tolerationSeconds: 3600
```

(1) **Exists** Operator は **value** を取りません。

この例では、テイントを、キー **key1**、値 **value1**、およびテイント effect **NoExecute** を持つ **node1** にテイントを配置します。

2. **テイントおよび容認コンポーネント** の表で説明されているパラメーターと共に以下のコマンドを使用してテイントをノードに追加します。

```
$ oc adm taint nodes <node_name> <key>=<value>:<effect>
```

以下に例を示します。

```
$ oc adm taint nodes node1 key1=value1:NoExecute
```

このコマンドは、キー **key1**、値 **value1**、および effect **NoExecute** を持つテイントを **node1** に配置します。

注記

NoSchedule テイントをコントロールプレーンノード (別名マスターノード) に追加する場合、ノードには、デフォルトで追加される **node-role.kubernetes.io/master=:NoSchedule** テイントが必要です。

以下に例を示します。

```
apiVersion: v1
kind: Node
metadata:
  annotations:
    machine.openshift.io/machine: openshift-machine-api/ci-ln-62s7gtb-f76d1-v8jxv-master-0
    machineconfiguration.openshift.io/currentConfig: rendered-master-cdc1ab7da414629332cc4c3926e6e59c
...
spec:
  taints:
  - effect: NoSchedule
    key: node-role.kubernetes.io/master
...
```

Pod の容認はノードのテイントに一致します。いずれかの容認のある Pod は **node1** にスケジュールできます。

3.7.2.1. マシンセットを使用したテイントおよび容認の追加

マシンセットを使用してテイントをノードに追加できます。**MachineSet** オブジェクトに関連付けられるすべてのノードがテイントで更新されます。容認は、ノードに直接追加されたテイントと同様に、マシンセットによって追加されるテイントに応答します。

手順

1. **Pod** 仕様を **tolerations** スタンザを含めるように編集して、容認を Pod に追加します。

   Equal 演算子を含む Pod 設定ファイルのサンプル

```
spec:
....
  template:
....
    spec:
      tolerations:
      - key: "key1" (1)
        value: "value1"
        operator: "Equal"
        effect: "NoExecute"
        tolerationSeconds: 3600 (2)
```

   (1) **テイントおよび容認コンポーネント** の表で説明されている toleration パラメーターです。

   (2) **tolerationSeconds** パラメーターは、エビクトする前に Pod をどの程度の期間ノードにバインドさせるかを指定します。

   以下に例を示します。

   Exists 演算子を含む Pod 設定ファイルのサンプル

```
spec:
  tolerations:
  - key: "key1"
    operator: "Exists"
    effect: "NoExecute"
    tolerationSeconds: 3600
```

2. テイントを **MachineSet** オブジェクトに追加します。

   a. テイントを付けるノードの **MachineSet** YAML を編集するか、または新規 **MachineSet** オブジェクトを作成できます。

```
$ oc edit machineset <machineset>
```

   b. テイントを **spec.template.spec** セクションに追加します。

   マシンセット仕様のテイントの例

```
spec:
....
  template:
....
```

```
    spec:
      taints:
      - effect: NoExecute
        key: key1
        value: value1
....
```

この例では、キー **key1**、値 **value1**、およびテイント effect **NoExecute** を持つテイントをノードに配置します。

c. マシンセットを 0 にスケールダウンします。

```
$ oc scale --replicas=0 machineset <machineset> -n openshift-machine-api
```

マシンが削除されるまで待機します。

d. マシンセットを随時スケールアップします。

```
$ oc scale --replicas=2 machineset <machineset> -n openshift-machine-api
```

マシンが起動するまで待ちます。テイントは **MachineSet** オブジェクトに関連付けられたノードに追加されます。

3.7.2.2. テイントおよび容認 (Toleration) 使ってユーザーをノードにバインドする

ノードのセットを特定のユーザーセットによる排他的な使用のために割り当てる必要がある場合、容認をそれらの Pod に追加します。次に、対応するテイントをそれらのノードに追加します。容認が設定された Pod は、テイントが付けられたノードまたはクラスター内の他のノードを使用できます。

Pod がテイントが付けられたノードのみにスケジュールされるようにするには、ラベルを同じノードセットに追加し、ノードのアフィニティーを Pod に追加し、Pod がそのラベルの付いたノードのみにスケジュールできるようにします。

手順

ノードをユーザーの使用可能な唯一のノードとして設定するには、以下を実行します。

1. 対応するテイントをそれらのノードに追加します。
   以下に例を示します。

```
$ oc adm taint nodes node1 dedicated=groupName:NoSchedule
```

2. カスタム受付コントローラーを作成して容認を Pod に追加します。

3.7.2.3. ノードセレクターおよび容認を使用したプロジェクトの作成

ノードセレクターおよび容認 (アノテーションとして設定されたもの) を使用するプロジェクトを作成して、Pod の特定のノードへの配置を制御できます。プロジェクトで作成された後続のリソースは、容認に一致するテイントを持つノードでスケジュールされます。

前提条件

- マシンセットを使用するか、またはノードを直接編集して、ノード選択のラベルが 1 つ以上のノードに追加されている。

- マシンセットを使用するか、またはノードを直接編集して、テイントが 1 つ以上のノードに追加されている。

手順

1. **metadata.annotations** セクションにノードセレクターおよび容認を指定して、**Project** リソース定義を作成します。

   project.yaml ファイルの例

```
kind: Project
apiVersion: project.openshift.io/v1
metadata:
  name: <project_name> 1
  annotations:
    openshift.io/node-selector: '<label>' 2
    scheduler.alpha.kubernetes.io/defaultTolerations: >-
      [{"operator": "Exists", "effect": "NoSchedule", "key":
      "<key_name>"} 3
      ]
```

   1. プロジェクト名。
   2. デフォルトのノードセレクターラベル。
   3. **テイントおよび容認コンポーネント** の表で説明されている toleration パラメーターです。この例では、**NoSchedule** の effect を使用します。これにより、ノード上の既存の Pod はそのまま残り、**Exists** Operator は値を取得しません。

2. **oc apply** コマンドを使用してプロジェクトを作成します。

```
$ oc apply -f project.yaml
```

<project_name> namespace で作成された後続のリソースは指定されたノードにスケジュールされます。

関連情報

- テイントおよび容認の追加を ノードに手動で実行、または マシンセットを使用する
- プロジェクトスコープのノードセレクターの作成
- Operator ワークロードの Pod の配置

3.7.2.4. テイントおよび容認 (Toleration) を使って特殊ハードウェアを持つノードを制御する

ノードの小規模なサブセットが特殊ハードウェアを持つクラスターでは、テイントおよび容認 (Toleration) を使用して、特殊ハードウェアを必要としない Pod をそれらのノードから切り離し、特殊ハードウェアを必要とする Pod をそのままにすることができます。また、特殊ハードウェアを必要とする Pod に対して特定のノードを使用することを要求することもできます。

これは、特殊ハードウェアを必要とする Pod に容認を追加し、特殊ハードウェアを持つノードにテイントを付けることで実行できます。

手順

特殊ハードウェアを持つノードが特定の Pod 用に予約されるようにするには、以下を実行します。

1. 容認を特別なハードウェアを必要とする Pod に追加します。
以下に例を示します。

```
spec:
  tolerations:
    - key: "disktype"
      value: "ssd"
      operator: "Equal"
      effect: "NoSchedule"
      tolerationSeconds: 3600
```

2. 以下のコマンドのいずれかを使用して、特殊ハードウェアを持つノードにテイントを設定します。

```
$ oc adm taint nodes <node-name> disktype=ssd:NoSchedule
```

または、以下を実行します。

```
$ oc adm taint nodes <node-name> disktype=ssd:PreferNoSchedule
```

3.7.3. テイントおよび容認 (Toleration) の削除

必要に応じてノードからテイントを、Pod から容認をそれぞれ削除できます。最初に容認を Pod に追加してからテイントをノードに追加して、容認を追加する前に Pod がノードから削除されないようにする必要があります。

手順

テイントおよび容認 (Toleration) を削除するには、以下を実行します。

1. ノードからテイントを削除するには、以下を実行します。

```
$ oc adm taint nodes <node-name> <key>-
```

以下に例を示します。

```
$ oc adm taint nodes ip-10-0-132-248.ec2.internal key1-
```

出力例

```
node/ip-10-0-132-248.ec2.internal untainted
```

2. Pod から容認を削除するには、容認を削除するための **Pod** 仕様を編集します。

```
spec:
  tolerations:
  - key: "key2"
```

```
operator: "Exists"
effect: "NoExecute"
tolerationSeconds: 3600
```

3.8. ノードセレクターの使用による特定ノードへの POD の配置

ノードセレクター は、ノードのカスタムラベルと Pod で指定されるセレクターを使用して定義されるキー/値のペアのマップを指定します。

Pod がノードで実行する要件を満たすには、Pod にはノードのラベルと同じキー/値のペアがなければなりません。

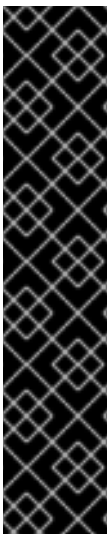
3.8.1. ノードセレクターについて

Pod でノードセレクターを使用し、ノードでラベルを使用して、Pod がスケジュールされる場所を制御できます。ノードセレクターにより、OpenShift Container Platform は一致するラベルが含まれるノード上に Pod をスケジュールします。

ノードセレクターを使用して特定の Pod を特定のノードに配置し、クラスタースコープのノードセレクターを使用して特定ノードの新規 Pod をクラスター内の任意の場所に配置し、プロジェクトノードを使用して新規 Pod を特定ノードのプロジェクトに配置できます。

たとえば、クラスター管理者は、作成するすべての Pod にノードセレクターを追加して、アプリケーション開発者が地理的に最も近い場所にあるノードにのみ Pod をデプロイできるインフラストラクチャーを作成できます。この例では、クラスターは 2 つのリージョンに分散する 5 つのデータセンターで設定されます。米国では、ノードに **us-east**、**us-central**、または **us-west** のラベルを付けます。アジア太平洋リージョン (APAC) では、ノードに **apac-east** または **apac-west** のラベルを付けます。開発者は、Pod がこれらのノードにスケジュールされるように、作成する Pod にノードセレクターを追加できます。

Pod オブジェクトにノードセレクターが含まれる場合でも、一致するラベルを持つノードがない場合、Pod はスケジュールされません。



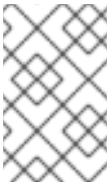
重要

同じ Pod 設定でノードセレクターとノードのアフィニティーを使用している場合は、以下のルールが Pod のノードへの配置を制御します。

- **nodeSelector** と **nodeAffinity** の両方を設定する場合、Pod が候補ノードでスケジュールされるにはどちらの条件も満たしている必要があります。
- **nodeAffinity** タイプに関連付けられた複数の **nodeSelectorTerms** を指定する場合、**nodeSelectorTerms** のいずれかが満たされている場合に Pod をノードにスケジュールすることができます。
- **nodeSelectorTerms** に関連付けられた複数の **matchExpressions** を指定する場合、すべての **matchExpressions** が満たされている場合にのみ Pod をノードにスケジュールすることができます。

特定の Pod およびノードのノードセレクター

ノードセレクターおよびラベルを使用して、特定の Pod がスケジュールされるノードを制御できます。
ノードセレクターおよびラベルを使用するには、まずノードにラベルを付けて Pod がスケジュール解除されないようにしてから、ノードセレクターを Pod に追加します。



注記

ノードセレクターを既存のスケジュールされている Pod に直接追加することはできません。デプロイメント設定などの Pod を制御するオブジェクトにラベルを付ける必要があります。

たとえば、以下の **Node** オブジェクトには **region: east** ラベルがあります。

ラベルを含む Node オブジェクトのサンプル

```
kind: Node
apiVersion: v1
metadata:
  name: ip-10-0-131-14.ec2.internal
  selfLink: /api/v1/nodes/ip-10-0-131-14.ec2.internal
  uid: 7bc2580a-8b8e-11e9-8e01-021ab4174c74
  resourceVersion: '478704'
  creationTimestamp: '2019-06-10T14:46:08Z'
  labels:
    kubernetes.io/os: linux
    failure-domain.beta.kubernetes.io/zone: us-east-1a
    node.openshift.io/os_version: '4.5'
    node-role.kubernetes.io/worker: ''
    failure-domain.beta.kubernetes.io/region: us-east-1
    node.openshift.io/os_id: rhcos
    beta.kubernetes.io/instance-type: m4.large
    kubernetes.io/hostname: ip-10-0-131-14
    beta.kubernetes.io/arch: amd64
    region: east (1)
```

(1) Pod ノードセレクターに一致するラベル。

Pod には **type: user-node,region: east** ノードセレクターがあります。

ノードセレクターが含まれる Pod オブジェクトのサンプル

```
apiVersion: v1
kind: Pod

....

spec:
  nodeSelector: (1)
    region: east
    type: user-node
```

(1) ノードトラベルに一致するノードセレクター。

サンプル Pod 仕様を使用して Pod を作成する場合、これはサンプルノードでスケジュールできます。

クラススコープのデフォルトノードセレクター

デフォルトのクラススコープのノードセレクターを使用する場合、クラスターで Pod を作成すると、OpenShift Container Platform はデフォルトのノードセレクターを Pod に追加し、一致するラベルのあるノードで Pod をスケジュールします。
たとえば、以下の **Scheduler** オブジェクトにはデフォルトのクラススコープの **region=east** および **type=user-node** ノードセレクターがあります。

スケジューラー Operator カスタムリソースの例

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  name: cluster
...

spec:
  defaultNodeSelector: type=user-node,region=east
...
```

クラスター内のノードには **type=user-node,region=east** ラベルがあります。

Node オブジェクトの例

```
apiVersion: v1
kind: Node
metadata:
  name: ci-ln-qg1il3k-f76d1-hlmhl-worker-b-df2s4
...
  labels:
    region: east
    type: user-node
...
```

ノードセレクターを持つ Pod オブジェクトの例

```
apiVersion: v1
kind: Pod
...

spec:
  nodeSelector:
    region: east
...
```

サンプルクラスターでサンプル Pod 仕様を使用して Pod を作成する場合、Pod はクラススコープのノードセレクターで作成され、ラベルが付けられたノードにスケジュールされます。

ラベルが付けられたノード上の Pod を含む Pod 一覧の例

```
NAME    READY   STATUS    RESTARTS   AGE   IP          NODE
NOMINATED NODE   READINESS GATES
pod-s1   1/1    Running   0          20s   10.131.2.6   ci-ln-qg1il3k-f76d1-hlmhl-worker-b-df2s4
<none>          <none>
```

注記

Pod を作成するプロジェクトにプロジェクトノードセレクターがある場合、そのセレクターはクラスタースコープのセレクターよりも優先されます。Pod にプロジェクトノードセレクターがない場合、Pod は作成されたり、スケジュールされたりしません。

プロジェクトノードセレクター

プロジェクトノードセレクターを使用する場合、このプロジェクトで Pod を作成すると、OpenShift Container Platform はノードセレクターを Pod に追加し、Pod を一致するラベルを持つノードでスケジュールします。クラスタースコープのデフォルトノードセレクターがない場合、プロジェクトノードセレクターが優先されます。
たとえば、以下のプロジェクトには **region=east** ノードセレクターがあります。

Namespace オブジェクトの例

```
apiVersion: v1
kind: Namespace
metadata:
  name: east-region
  annotations:
    openshift.io/node-selector: "region=east"
...
```

以下のノードには **type=user-node,region=east** ラベルがあります。

Node オブジェクトの例

```
apiVersion: v1
kind: Node
metadata:
  name: ci-ln-qg1il3k-f76d1-hlmhl-worker-b-df2s4
...
  labels:
    region: east
    type: user-node
...
```

Pod をこのサンプルプロジェクトでサンプル Pod 仕様を使用して作成する場合、Pod はプロジェクトノードセレクターで作成され、ラベルが付けられたノードにスケジュールされます。

Pod オブジェクトの例

```
apiVersion: v1
kind: Pod
metadata:
  namespace: east-region
...
spec:
  nodeSelector:
    region: east
    type: user-node
...
```

ラベルが付けられたノード上の Pod を含む Pod 一覧の例

```
NAME    READY   STATUS    RESTARTS   AGE   IP           NODE
NOMINATED NODE   READINESS GATES
pod-s1   1/1     Running   0          20s   10.131.2.6   ci-ln-qg1il3k-f76d1-hlmhl-worker-b-df2s4
<none>           <none>
```

Pod に異なるノードセレクターが含まれる場合、プロジェクトの Pod は作成またはスケジュールされません。たとえば、以下の Pod をサンプルプロジェクトにデプロイする場合、これは作成されません。

無効なノードセレクターを持つ Pod オブジェクトの例

```
apiVersion: v1
kind: Pod
...

spec:
  nodeSelector:
    region: west

....
```

3.8.2. ノードセレクターの使用による Pod 配置の制御

Pod でノードセレクターを使用し、ノードでラベルを使用して、Pod がスケジュールされる場所を制御できます。ノードセレクターにより、OpenShift Container Platform は一致するラベルが含まれるノード上に Pod をスケジュールします。

ラベルをノード、マシンセット、またはマシン設定に追加します。マシンセットにラベルを追加すると、ノードまたはマシンが停止した場合に、新規ノードにそのラベルが追加されます。ノードまたはマシン設定に追加されるラベルは、ノードまたはマシンが停止すると維持されません。

ノードセレクターを既存 Pod に追加するには、ノードセレクターを **ReplicaSet** オブジェクト、**DaemonSet** オブジェクト、**StatefulSet** オブジェクト、**Deployment** オブジェクト、または **DeploymentConfig** オブジェクトなどの Pod の制御オブジェクトに追加します。制御オブジェクト下の既存 Pod は、一致するラベルを持つノードで再作成されます。新規 Pod を作成する場合、ノードセレクターを **Pod** 仕様に直接追加できます。

注記

ノードセレクターを既存のスケジュールされている Pod に直接追加することはできません。

前提条件

ノードセレクターを既存 Pod に追加するには、Pod の制御オブジェクトを判別します。たとえば、**router-default-66d5cf9464-m2g75** Pod は **router-default-66d5cf9464** レプリカセットによって制御されます。

```
$ oc describe pod router-default-66d5cf9464-7pwkc

Name:             router-default-66d5cf9464-7pwkc
```

```
Namespace:          openshift-ingress

....

Controlled By:      ReplicaSet/router-default-66d5cf9464
```

Web コンソールでは、Pod YAML の **ownerReferences** に制御オブジェクトを一覧表示します。

```
ownerReferences:
  - apiVersion: apps/v1
    kind: ReplicaSet
    name: router-default-66d5cf9464
    uid: d81dd094-da26-11e9-a48a-128e7edf0312
    controller: true
    blockOwnerDeletion: true
```

手順

1. マシンセットを使用するか、またはノードを直接編集してラベルをノードに追加します。
   - **MachineSet** オブジェクトを使用して、ノードの作成時にマシンセットによって管理されるノードにラベルを追加します。
     a. 以下のコマンドを実行してラベルを **MachineSet** オブジェクトに追加します。

     ```
     $ oc patch MachineSet <name> --type='json' -
     p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"<key>="
     <value>","<key>"="<value>"}}]'  -n openshift-machine-api
     ```

     以下に例を示します。

     ```
     $ oc patch MachineSet abc612-msrtw-worker-us-east-1c  --type='json' -
     p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
     node","region":"east"}}]'  -n openshift-machine-api
     ```

     b. **oc edit** コマンドを使用して、ラベルが **MachineSet** オブジェクトに追加されていることを確認します。
     以下に例を示します。

     ```
     $ oc edit MachineSet abc612-msrtw-worker-us-east-1c -n openshift-machine-api
     ```

     MachineSet オブジェクトの例

     ```
     apiVersion: machine.openshift.io/v1beta1
     kind: MachineSet

     ....

     spec:
     ...
       template:
         metadata:
     ...
         spec:
     ```

```
      metadata:
        labels:
          region: east
          type: user-node
....
```

- ラベルをノードに直接追加します。

  a. ノードの **Node** オブジェクトを編集します。

  ```
  $ oc label nodes <name> <key>=<value>
  ```

  たとえば、ノードにラベルを付けるには、以下を実行します。

  ```
  $ oc label nodes ip-10-0-142-25.ec2.internal type=user-node region=east
  ```

  b. ラベルがノードに追加されていることを確認します。

  ```
  $ oc get nodes -l type=user-node,region=east
  ```

  出力例

  ```
  NAME                          STATUS   ROLES    AGE   VERSION
  ip-10-0-142-25.ec2.internal   Ready    worker   17m   v1.18.3+002a51f
  ```

2. 一致するノードセレクターを Pod に追加します。

- ノードセレクターを既存 Pod および新規 Pod に追加するには、ノードセレクターを Pod の制御オブジェクトに追加します。

  ラベルを含む ReplicaSet オブジェクトのサンプル

```
kind: ReplicaSet

....

spec:

....

  template:
    metadata:
      creationTimestamp: null
      labels:
        ingresscontroller.operator.openshift.io/deployment-ingresscontroller: default
        pod-template-hash: 66d5cf9464
    spec:
      nodeSelector:
        kubernetes.io/os: linux
        node-role.kubernetes.io/worker: ''
        type: user-node 1
```

1 ノードセレクターを追加します。

- ノードセレクターを特定の新規 Pod に追加するには、セレクターを **Pod** オブジェクトに直接追加します。

ノードセレクターを持つ Pod オブジェクトの例

```
apiVersion: v1
kind: Pod

....

spec:
  nodeSelector:
    region: east
    type: user-node
```

注記

ノードセレクターを既存のスケジュールされている Pod に直接追加することはできません。

3.8.3. クラススコープのデフォルトノードセレクターの作成

クラスター内の作成されたすべての Pod を特定のノードに制限するために、デフォルトのクラススコープのノードセレクターをノード上のラベルと共に Pod で使用することができます。

クラススコープのノードセレクターを使用する場合、クラスターで Pod を作成すると、OpenShift Container Platform はデフォルトのノードセレクターを Pod に追加し、一致するラベルのあるノードで Pod をスケジュールします。

スケジューラー Operator カスタムリソース (CR) を編集して、クラススコープのノードセレクターを設定します。ラベルをノード、マシンセット、またはマシン設定に追加します。マシンセットにラベルを追加すると、ノードまたはマシンが停止した場合に、新規ノードにそのラベルが追加されます。ノードまたはマシン設定に追加されるラベルは、ノードまたはマシンが停止すると維持されません。

注記

Pod にキーと値のペアを追加できます。ただし、デフォルトキーの異なる値を追加することはできません。

手順

デフォルトのクラススコープのセレクターを追加するには、以下を実行します。

1. スケジューラー Operator CR を編集して、デフォルトのクラススコープのノードクラスターを追加します。

```
$ oc edit scheduler cluster
```

ノードセレクターを含むスケジューラー Operator CR のサンプル

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  name: cluster
```

```
...

spec:
  defaultNodeSelector: type=user-node,region=east (1)
  mastersSchedulable: false
  policy:
    name: ""
```

(1) 適切な **<key>:<value>** ペアが設定されたノードセレクターを追加します。

この変更を加えた後に、**openshift-kube-apiserver** プロジェクトの Pod の再デプロイを待機します。これには数分の時間がかかる場合があります。デフォルトのクラスター全体のノードセレクターは、Pod の再起動まで有効になりません。

2. マシンセットを使用するか、またはノードを直接編集してラベルをノードに追加します。

- マシンセットを使用して、ノードの作成時にマシンセットによって管理されるノードにラベルを追加します。

a. 以下のコマンドを実行してラベルを **MachineSet** オブジェクトに追加します。

```
$ oc patch MachineSet <name> --type='json' -
p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"<key>="
<value>","<key>"="<value>"}}]'  -n openshift-machine-api (1)
```

(1) それぞれのラベルに **<key>** /**<value>** ペアを追加します。

以下に例を示します。

```
$ oc patch MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c --type='json' -
p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
node","region":"east"}}]'  -n openshift-machine-api
```

b. **oc edit** コマンドを使用して、ラベルが **MachineSet** オブジェクトに追加されていることを確認します。
以下に例を示します。

```
$ oc edit MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

出力例

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
...
spec:
...
  template:
    metadata:
...
      spec:
        metadata:
```

```
      labels:
        region: east
        type: user-node
```

c. **0** にスケールダウンし、ノードをスケールアップして、そのマシンセットに関連付けられたノードを再デプロイします。
以下に例を示します。

```
$ oc scale --replicas=0 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

```
$ oc scale --replicas=1 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

d. ノードの準備ができ、利用可能な状態になったら、**oc get** コマンドを使用してラベルがノードに追加されていることを確認します。

```
$ oc get nodes -l <key>=<value>
```

以下に例を示します。

```
$ oc get nodes -l type=user-node
```

出力例

```
NAME                                         STATUS   ROLES    AGE   VERSION
ci-ln-l8nry52-f76d1-hl7m7-worker-c-vmqzp   Ready    worker   61s   v1.18.3+002a51f
```

- ラベルをノードに直接追加します。

a. ノードの **Node** オブジェクトを編集します。

```
$ oc label nodes <name> <key>=<value>
```

たとえば、ノードにラベルを付けるには、以下を実行します。

```
$ oc label nodes ci-ln-l8nry52-f76d1-hl7m7-worker-b-tgq49 type=user-node region=east
```

b. **oc get** コマンドを使用して、ラベルがノードに追加されていることを確認します。

```
$ oc get nodes -l <key>=<value>,<key>=<value>
```

以下に例を示します。

```
$ oc get nodes -l type=user-node,region=east
```

出力例

```
NAME                                         STATUS   ROLES    AGE   VERSION
ci-ln-l8nry52-f76d1-hl7m7-worker-b-tgq49   Ready    worker   17m   v1.18.3+002a51f
```

3.8.4. プロジェクトスコープのノードセレクターの作成

プロジェクトで作成されたすべての Pod をラベルが付けられたノードに制限するために、プロジェクトのノードセレクターをノード上のラベルと共に使用できます。

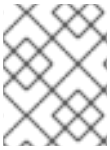
このプロジェクトで Pod を作成する場合、OpenShift Container Platform はノードセレクターをプロジェクトの Pod に追加し、プロジェクトの一致するラベルを持つノードで Pod をスケジュールします。クラスタースコープのデフォルトノードセレクターがない場合、プロジェクトノードセレクターが優先されます。

You add node selectors to a project by editing the **Namespace** object to add the **openshift.io/node-selector** parameter.ラベルをノード、マシンセット、またはマシン設定に追加します。マシンセットにラベルを追加すると、ノードまたはマシンが停止した場合に、新規ノードにそのラベルが追加されます。ノードまたはマシン設定に追加されるラベルは、ノードまたはマシンが停止すると維持されません。

Pod オブジェクトにノードセレクターが含まれる場合でも、一致するノードセレクターを持つプロジェクトがない場合、Pod はスケジュールされません。その仕様から Pod を作成すると、以下のメッセージと同様のエラーが表示されます。

エラーメッセージの例

```
Error from server (Forbidden): error when creating "pod.yaml": pods "pod-4" is forbidden: pod node label selector conflicts with its project node label selector
```



注記

Pod にキーと値のペアを追加できます。ただし、プロジェクトキーに異なる値を追加することはできません。

手順

デフォルトのプロジェクトノードセレクターを追加するには、以下を実行します。

1. namespace を作成するか、既存の namespace を編集して **openshift.io/node-selector** パラメーターを追加します。

```
$ oc edit namespace <name>
```

出力例

```
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    openshift.io/node-selector: "type=user-node,region=east" 1
    openshift.io/description: ""
    openshift.io/display-name: ""
    openshift.io/requester: kube:admin
    openshift.io/sa.scc.mcs: s0:c30,c5
    openshift.io/sa.scc.supplemental-groups: 1000880000/10000
    openshift.io/sa.scc.uid-range: 1000880000/10000
  creationTimestamp: "2021-05-10T12:35:04Z"
  labels:
```

```
    kubernetes.io/metadata.name: demo
  name: demo
  resourceVersion: "145537"
  uid: 3f8786e3-1fcb-42e3-a0e3-e2ac54d15001
spec:
  finalizers:
  - kubernetes
```

1 適切な **<key>:<value>** ペアを持つ **openshift.io/node-selector** を追加します。

2. マシンセットを使用するか、またはノードを直接編集してラベルをノードに追加します。

- **MachineSet** オブジェクトを使用して、ノードの作成時にマシンセットによって管理されるノードにラベルを追加します。

a. 以下のコマンドを実行してラベルを **MachineSet** オブジェクトに追加します。

```
$ oc patch MachineSet <name> --type='json' -
p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"<key>="
<value>","<key>"="<value>"}}]'  -n openshift-machine-api
```

以下に例を示します。

```
$ oc patch MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c --type='json' -
p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
node","region":"east"}}]'  -n openshift-machine-api
```

b. **oc edit** コマンドを使用して、ラベルが **MachineSet** オブジェクトに追加されていることを確認します。
以下に例を示します。

```
$ oc edit MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

出力例

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
...
spec:
...
  template:
    metadata:
...
    spec:
      metadata:
        labels:
          region: east
          type: user-node
```

c. そのマシンセットに関連付けられたノードを再デプロイします。
以下に例を示します。

```
$ oc scale --replicas=0 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

```
$ oc scale --replicas=1 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

d. ノードの準備ができ、利用可能な状態になったら、**oc get** コマンドを使用してラベルがノードに追加されていることを確認します。

```
$ oc get nodes -l <key>=<value>
```

以下に例を示します。

```
$ oc get nodes -l type=user-node,region=east
```

出力例

```
NAME                                STATUS   ROLES    AGE   VERSION
ci-ln-l8nry52-f76d1-hl7m7-worker-c-vmqzp   Ready    worker   61s   v1.18.3+002a51f
```

- ラベルをノードに直接追加します。

a. **Node** オブジェクトを編集してラベルを追加します。

```
$ oc label <resource> <name> <key>=<value>
```

たとえば、ノードにラベルを付けるには、以下を実行します。

```
$ oc label nodes ci-ln-l8nry52-f76d1-hl7m7-worker-c-tgq49 type=user-node region=east
```

b. **oc get** コマンドを使用して、ラベルが **Node** オブジェクトに追加されていることを確認します。

```
$ oc get nodes -l <key>=<value>
```

以下に例を示します。

```
$ oc get nodes -l type=user-node,region=east
```

出力例

```
NAME                                STATUS   ROLES    AGE   VERSION
ci-ln-l8nry52-f76d1-hl7m7-worker-b-tgq49   Ready    worker   17m   v1.18.3+002a51f
```

関連情報

- ノードセレクターおよび容認を使用したプロジェクトの作成

3.9. POD トポロジー分散制約を使用した POD 配置の制御

Pod トポロジー分散制約を使用して、ノード、ゾーン、リージョンその他のユーザー定義のトポロジードメイン間で Pod の配置を制御できます。

3.9.1. Pod トポロジー分散制約について

Pod トポロジー分散制約 を使用することで、障害ドメイン全体にまたがる Pod の分散に対する詳細な制御を実現し、高可用性とより効率的なリソースの使用を実現できます。

OpenShift Container Platform 管理者はノードにラベルを付け、リージョン、ゾーン、ノード、他のユーザー定義ドメインなどのトポロジー情報を提供できます。これらのラベルをノードに設定した後に、ユーザーは Pod トポロジーの分散制約を定義し、これらのトポロジードメイン全体での Pod の配置を制御できます。

グループ化する Pod を指定し、それらの Pod が分散されるトポロジードメインと、許可できるスキューを指定します。制約により、分散される際に同じ namespace 内の Pod のみが一致し、グループ化されます。

3.9.2. Pod トポロジー分散制約の設定

以下の手順は、Pod トポロジー分散制約を、ゾーンに基づいて指定されたラベルに一致する Pod を分散するように設定する方法を示しています。

複数の Pod トポロジー分散制約を指定できますが、それらが互いに競合しないようにする必要があります。Pod を配置するには、すべての Pod トポロジー分散制約を満たしている必要があります。

前提条件

- クラスター管理者は、必要なラベルをノードに追加している。

手順

1. **Pod** 仕様を作成し、Pod トポロジーの分散制約を指定します。

   pod-spec.yaml ファイルの例

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    foo: bar
spec:
  topologySpreadConstraints:
  - maxSkew: 1 (1)
    topologyKey: topology.kubernetes.io/zone (2)
    whenUnsatisfiable: DoNotSchedule (3)
    labelSelector: (4)
      matchLabels:
        foo: bar (5)
  containers:
  - image: "docker.io/ocpqe/hello-pod"
    name: hello-pod
```

(1) 任意の 2 つのトポロジードメイン間の Pod 数の最大差。デフォルトは **1** で、**0** の値を指定

することはできません。

2 ノードラベルのキー。このキーと同じ値を持つノードは同じトポロジーにあると見なされます。

3 分散制約を満たさない場合に Pod を処理する方法です。デフォルトは **DoNotSchedule** であり、これはスケジューラーに Pod をスケジュールしないように指示します。**ScheduleAnyway** に設定して Pod を依然としてスケジュールできますが、スケジューラーはクラスターがさらに不均衡な状態になるのを防ぐためにスキューの適用を優先します。

4 制約を満たすために、分散される際に、このラベルセレクターに一致する Pod はグループとしてカウントされ、認識されます。ラベルセレクターを指定してください。指定しないと、Pod が一致しません。

5 今後適切にカウントされるようにするには、この **Pod** 仕様がこのラベルセレクターに一致するようにラベルを設定していることも確認してください。

2. Pod を作成します。

```
$ oc create -f pod-spec.yaml
```

3.9.3. Pod トポロジー分散制約の例

以下の例は、Pod トポロジー設定分散制約の設定を示しています。

3.9.3.1. 単一 Pod トポロジー分散制約の例

このサンプル **Pod** 仕様は単一の Pod トポロジー分散制約を定義します。これは **foo:bar** というラベルが付いた Pod で一致し、ゾーン間で分散され、スキューの **1** を指定し、これらの要件を満たさない場合に Pod をスケジュールしません。

```
kind: Pod
apiVersion: v1
metadata:
  name: my-pod
  labels:
    foo: bar
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  containers:
  - image: "docker.io/ocpqe/hello-pod"
    name: hello-pod
```

3.9.3.2. 複数の Pod トポロジー分散制約の例

このサンプル **Pod** 仕様は 2 つの Pod トポロジー分散制約を定義します。どちらの場合も **foo:bar** というラベルが付けられた Pod で一致し、スキューの **1** を指定し、これらの要件を満たしていない Pod をスケジュールしません。

最初の制約は、ユーザー定義ラベルの **node** に基づいて Pod を分散し、2 つ目の制約はユーザー定義ラベルの **rack** に基づいて Pod を分散します。Pod がスケジュールされるには、両方の制約を満たす必要があります。

```
kind: Pod
apiVersion: v1
metadata:
  name: my-pod-2
  labels:
    foo: bar
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: node
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  - maxSkew: 1
    topologyKey: rack
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  containers:
  - image: "docker.io/ocpqe/hello-pod"
    name: hello-pod
```

3.9.4. 関連情報

- ノードでラベルを更新する方法について

3.10. カスタムスケジューラーの実行

デフォルトのスケジューラーと共に複数のカスタムスケジューラーを実行し、各 Pod に使用するスケジューラーを設定できます。

重要

これは OpenShift Container Platform でカスタムスケジューラーを使用することはサポートされていますが、Red Hat ではカスタムスケジューラーの機能を直接サポートしません。

デフォルトのスケジューラーを設定する方法については、Configuring the default scheduler to control pod placement を参照してください。

特定のスケジューラーを使用して指定された Pod をスケジュールするには、**Pod** の仕様にスケジューラーの名前を指定 します。

3.10.1. カスタムスケジューラーのデプロイ

クラスターにカスタムスケジューラーを追加するには、デプロイメントにカスタムスケジューラーのイメージを追加します。

前提条件

- **cluster-admin** ロールを持つユーザーとしてクラスターにアクセスできる。
- スケジューラーバイナリーがある。

注記

スケジューラーバイナリーの作成方法に関する情報は、本書では扱っておりません。たとえば、Kubernetes ドキュメントの Configure Multiple Schedulers を参照してください。カスタムスケジューラーの実際の機能は、Red Hat ではサポートされない点に留意してください。

- スケジューラーバイナリーを含むイメージを作成し、これをレジストリーにプッシュしている。

手順

1. カスタムスケジューラーのデプロイメントリソースを含むファイルを作成します。

   custom-scheduler.yaml ファイルの例

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: custom-scheduler
  namespace: kube-system 1
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: custom-scheduler-as-kube-scheduler
subjects:
- kind: ServiceAccount
  name: custom-scheduler
  namespace: kube-system 2
roleRef:
  kind: ClusterRole
  name: system:kube-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: custom-scheduler-as-volume-scheduler
subjects:
- kind: ServiceAccount
  name: custom-scheduler
  namespace: kube-system 3
roleRef:
  kind: ClusterRole
```

```
  name: system:volume-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    component: scheduler
    tier: control-plane
  name: custom-scheduler
  namespace: kube-system 4
spec:
  selector:
    matchLabels:
      component: scheduler
      tier: control-plane
  replicas: 1
  template:
    metadata:
      labels:
        component: scheduler
        tier: control-plane
        version: second
    spec:
      serviceAccountName: custom-scheduler
      containers:
      - command:
        - /usr/local/bin/kube-scheduler
        - --address=0.0.0.0
        - --leader-elect=false
        - --scheduler-name=custom-scheduler 5
        image: "<namespace>/<image_name>:<tag>" 6
        livenessProbe:
          httpGet:
            path: /healthz
            port: 10251
          initialDelaySeconds: 15
        name: kube-second-scheduler
        readinessProbe:
          httpGet:
            path: /healthz
            port: 10251
        resources:
          requests:
            cpu: '0.1'
        securityContext:
          privileged: false
        volumeMounts: []
      hostNetwork: false
      hostPID: false
      volumes: []
```

1 2 3 4 この手順では、**kube-system** namespace を使用しますが、お好みの namespace を使用することができます。

5 カスタムスケジューラーのコマンドには、異なる引数が必要な場合があります。たとえば、**--config** 引数を使用して、設定をマウントされたボリュームとして渡すことができま

6 カスタムスケジューラー用に作成したコンテナーイメージを指定します。

2. クラスター内にデプロイメントリソースを作成します。

```
$ oc create -f custom-scheduler.yaml
```

検証

- スケジューラー Pod が実行されていることを確認します。

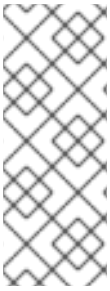
```
$ oc get pods -n kube-system
```

カスタムスケジューラー Pod は **Running** として一覧表示されます。

```
NAME                                     READY   STATUS    RESTARTS   AGE
custom-scheduler-6cd7c4b8bc-854zb                1/1     Running   0          2m
```

3.10.2. カスタムスケジューラーを使用した Pod のデプロイ

カスタムスケジューラーをクラスターにデプロイした後、デフォルトのスケジューラーではなくそのスケジューラーを使用するように Pod を設定できます。



注記

各スケジューラーには、クラスター内のリソースの個別のビューがあります。このため、各スケジューラーは独自のノードセットを動作する必要があります。

2 つ以上のスケジューラーが同じノードで動作する場合、それらは互いに介入し、利用可能なリソースよりも多くの Pod を同じノードにスケジュールする可能性があります。この場合、Pod はリソースが十分にないために拒否される可能性があります。

前提条件

- **cluster-admin** ロールを持つユーザーとしてクラスターにアクセスできる。
- カスタムスケジューラーがクラスターにデプロイされている。

手順

1. クラスターがロールベースアクセス制御 (RBAC) を使用する場合は、カスタムスケジューラー名を **system:kube-scheduler** クラスターロールに追加します。

   a. **system:kube-scheduler** クラスターロールを編集します。

   ```
   $ oc edit clusterrole system:kube-scheduler
   ```

   b. カスタムスケジューラーの名前を、**leases** および **endpoints** リソースの **resourceNames** 一覧に追加します。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: "2021-07-07T10:19:14Z"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:kube-scheduler
  resourceVersion: "125"
  uid: 53896c70-b332-420a-b2a4-f72c822313f2
rules:
...
- apiGroups:
  - coordination.k8s.io
  resources:
  - leases
  verbs:
  - create
- apiGroups:
  - coordination.k8s.io
  resourceNames:
  - kube-scheduler
  - custom-scheduler (1)
  resources:
  - leases
  verbs:
  - get
  - update
- apiGroups:
  - ""
  resources:
  - endpoints
  verbs:
  - create
- apiGroups:
  - ""
  resourceNames:
  - kube-scheduler
  - custom-scheduler (2)
  resources:
  - endpoints
  verbs:
  - get
  - update
...
```

(1) (2) この例では、**custom-scheduler** をカスタムスケジューラー名として使用します。

2. **Pod** 設定を作成し、**schedulerName** パラメーターでカスタムスケジューラーの名前を指定します。

custom-scheduler-example.yaml ファイルの例

```
apiVersion: v1
kind: Pod
metadata:
  name: custom-scheduler-example
  labels:
    name: custom-scheduler-example
spec:
  schedulerName: custom-scheduler 1
  containers:
  - name: pod-with-second-annotation-container
    image: docker.io/ocpqe/hello-pod
```

1 使用するカスタムスケジューラーの名前です。この例では **custom-scheduler** になります。スケジューラー名が指定されていない場合、Pod はデフォルトのスケジューラーを使用して自動的にスケジュールされます。

3. Pod を作成します。

```
$ oc create -f custom-scheduler-example.yaml
```

検証

1. 以下のコマンドを入力し、Pod が作成されたことを確認します。

```
$ oc get pod custom-scheduler-example
```

custom-scheduler-example Pod が出力に表示されます。

```
NAME                      READY     STATUS    RESTARTS   AGE
custom-scheduler-example   1/1       Running   0          4m
```

2. 以下のコマンドを入力し、カスタムスケジューラーが Pod をスケジュールしたことを確認します。

```
$ oc describe pod custom-scheduler-example
```

以下の切り捨てられた出力に示されるように、スケジューラー **custom-scheduler** が一覧表示されます。

```
Events:
  Type    Reason          Age       From                                  Message
  ----    ------          ----      ----                                  -------
  Normal  Scheduled       <unknown>  custom-scheduler                     Successfully
assigned default/custom-scheduler-example to <node_name>
```

3.10.3. 関連情報

- コンテナーのベストプラクティスについて

3.11. DESCHEDULER を使用した POD のエビクト

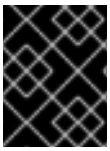
スケジューラー は新規 Pod をホストするのに最適なノードを判別するために使用されますが、Descheduler は実行中の Pod をエビクトするために使用され、Pod がより適したノードに再スケジュールされるようにできます。

3.11.1. Descheduler について

Descheduler を使用して Pod を特定のストラテジーに基づいてエビクトし、Pod がより適切なノードに再スケジュールされるようにできます。

以下のような状況では、実行中の Pod のスケジュールを解除することに利点があります。

- ノードの使用率が低くなっているか、使用率が高くなっている。
- テイントまたはラベルなどの、Pod およびノードアフィニティーの各種要件が変更され、当初のスケジュールの意思決定が特定のノードに適さなくなっている。
- ノードの障害により、Pod を移動する必要がある。
- 新規ノードがクラスターに追加されている。
- Pod が再起動された回数が多すぎる。



重要

Descheduler はエビクトされた Pod の置き換えをスケジュールしません。スケジューラーは、エビクトされた Pod に対してこのタスクを自動的に実行します。

Descheduler がノードから Pod をエビクトすることを決定する際には、以下の一般的なメカニズムを使用します。

- **openshift-*** および **kube-system** namespace の Pod はエビクトされることがありません。
- **priorityClassName** が **system-cluster-critical** または **system-node-critical** に設定されている Critical Pod はエビクトされることがありません。
- レプリケーションコントローラー、レプリカセット、デプロイメント、またはジョブの一部ではない静的な Pod、ミラーリングされた Pod、またはスタンドアロンの Pod は、再作成されないためにエビクトされません。
- デーモンセットに関連付けられた Pod はエビクトされることがありません。
- ローカルストレージを持つ Pod はエビクトされることがありません。
- Best effort Pod は、Burstable および Guaranteed Pod の前にエビクトされます。
- **descheduler.alpha.kubernetes.io/evict** アノテーションを持つすべてのタイプの Pod はエビクトの対象になります。このアノテーションはエビクションを防ぐチェックを上書きするために使用され、ユーザーはエビクトする Pod を選択できます。ユーザーは、Pod を再作成する方法と、Pod が再作成されるかどうかを認識している必要があります。
- Pod の Disruption Budget (PDB) が適用される Pod は、スケジュール解除が PDB に違反する場合にはエビクトされません。Pod は、エビクションサブリソースを使用して PDB を処理することでエビクトされます。

3.11.2. Descheduler プロファイル

以下の Descheduler ストラテジーを利用できます。

AffinityAndTaints

このプロファイルは、Pod 間の非アフィニティー、ノードアフィニティー、およびノードのテイントに違反する Pod をエビクトします。
これにより、以下のストラテジーが有効になります。

- **RemovePodsViolatingInterPodAntiAffinity**: Pod 間の非アフィニティーに違反する Pod を削除します。
- **RemovePodsViolatingNodeAffinity**: ノードのアフィニティー に違反する Pod を削除します。
- **RemovePodsViolatingNodeTaints**: ノード上の **NoSchedule** テイントに違反する Pod を削除します。
  ノードのアフィニティータイプが **requiredDuringSchedulingIgnoredDuringExecution** の Pod は削除されます。

TopologyAndDuplicates

このプロファイルは、ノード間で同様の Pod または同じトポロジードメインの Pod を均等に分散できるように Pod をエビクトします。
これにより、以下のストラテジーが有効になります。

- **RemovePodsViolatingTopologySpreadConstraint**: 均等に分散されていないとポロジードメインを見つけ、**DoNotSchedule** 制約を違反している場合により大きなものから Pod のエビクトを試行します。
- **RemoveDuplicates**: 1 つの Pod のみが同じノードで実行されているレプリカセット、 レプリケーションコントローラー、デプロイメントまたはジョブに関連付けられます。追加の Pod がある場合、それらの重複 Pod はクラスターに Pod を効果的に分散できるようにエビクトされます。

LifecycleAndUtilization

このプロファイルは長時間実行される Pod をエビクトし、ノード間のリソース使用状況のバランスを取ります。
これにより、以下のストラテジーが有効になります。

- **RemovePodsHavingTooManyRestarts** : コンテナーが何度も再起動された Pod を削除します。
  すべてのコンテナー (Init コンテナーを含む) での再起動の合計が 100 を超える Pod。
- **LowNodeUtilization**: 使用率の低いノードを検出し、可能な場合は過剰に使用されているノードから Pod をエビクトし、エビクトされた Pod の再作成がそれらの使用率の低いノードでスケジュールされるようにします。
  ノードは、使用率がすべてしきい値 (CPU、メモリー、Pod の数) について 20% 未満の場合に使用率が低いと見なされます。

  ノードは、使用率がすべてのしきい値 (CPU、メモリー、Pod の数) について 50% を超える場合に過剰に使用されていると見なされます。
- **PodLifeTime**: 古くなり過ぎた Pod をエビクトします。
  24 時間を経過した Pod は削除されます。

3.11.3. Descheduler のインストール

Descheduler はデフォルトで利用できません。Descheduler を有効にするには、Kube Descheduler Operator を OperatorHub からインストールし、1 つ以上の Descheduler プロファイルを有効にする必要があります。

前提条件

- クラスター管理者の権限。
- OpenShift Container Platform Web コンソールにアクセスします。

手順

1. OpenShift Container Platform Web コンソールにログインします。
2. Kube Descheduler Operator に必要な namespace を作成します。
   a. **Administration → Namespaces** に移動し、**Create Namespace** をクリックします。
   b. **Name** フィールドに **openshift-kube-descheduler-operator** を入力し、**Create** をクリックします。
3. Kube Descheduler Operator をインストールします。
   a. **Operators → OperatorHub** に移動します。
   b. **Kube Descheduler Operator** をフィルターボックスに入力します。
   c. **Kube Descheduler Operator** を選択し、**Install** をクリックします。
   d. **Install Operator** ページで、**A specific namespace on the cluster**を選択します。ドロップダウンメニューから **openshift-kube-descheduler-operator** を選択します。
   e. **Update Channel** および **Approval Strategy** の値を必要な値に調整します。
   f. **Install** をクリックします。
4. Descheduler インスタンスを作成します。
   a. **Operators → Installed Operators** ページから、 **Kube Descheduler Operator** をクリックします。
   b. **Kube Descheduler** タブを選択し、**Create KubeDescheduler**をクリックします。
   c. 必要に応じて設定を編集します。
      i. **Profiles** セクションを展開し、1 つ以上のプロファイルを選択して有効にします。**AffinityAndTaints** プロファイルはデフォルトで有効になっています。**Add Profile** をクリックして、追加のプロファイルを選択します。
      ii. オプション: **Descheduling Interval Seconds** フィールドを使用して、Descheduler の実行間の秒数を変更します。デフォルトは **3600** 秒です。
   d. **Create** をクリックします。

また、後で OpenShift CLI (**oc**) を使用して、Descheduler のプロファイルおよび設定を設定することもできます。Web コンソールから Descheduler インスタンスを作成する際にプロファイルを調整しない場合、**AffinityAndTaints** プロファイルはデフォルトで有効にされます。

3.11.4. Descheduler プロファイルの設定

Descheduler が Pod のエビクトに使用するプロファイルを設定できます。

前提条件

- クラスター管理者の権限

手順

1. **KubeDescheduler** オブジェクトを編集します。

```
$ oc edit kubedeschedulers.operator.openshift.io cluster -n openshift-kube-descheduler-operator
```

2. **spec.profiles** セクションに 1 つ以上のプロファイルを指定します。

```
apiVersion: operator.openshift.io/v1beta1
kind: KubeDescheduler
metadata:
  name: cluster
  namespace: openshift-kube-descheduler-operator
spec:
  deschedulingIntervalSeconds: 3600
  logLevel: Normal
  managementState: Managed
  operatorLogLevel: Normal
  profiles:
  - AffinityAndTaints      (1)
  - TopologyAndDuplicates  (2)
  - LifecycleAndUtilization (3)
```

(1) Pod 間の非アフィニティー、ノードアフィニティー、およびノードのテイントに違反する Pod をエビクトする **AffinityAndTaints** プロファイルを有効にします。

(2) ノード間で同様の Pod または同じトポロジードメインの Pod を均等に分散できるように Pod をエビクトする **TopologyAndDuplicates** プロファイルを有効にします。

(3) 長時間実行される Pod をエビクトし、ノード間のリソース使用状況のバランスを取る、**LifecycleAndUtilization** プロファイルを有効にします。

複数のプロファイルを有効にすることができますが、プロファイルを指定する順番は重要ではありません。

3. 変更を適用するためにファイルを保存します。

3.11.5. Descheduler の間隔の設定

Descheduler の実行間隔を設定できます。デフォルトは 3600 秒 (1 時間) です。

前提条件

- クラスター管理者の権限

手順

1. **KubeDescheduler** オブジェクトを編集します。

```
$ oc edit kubedeschedulers.operator.openshift.io cluster -n openshift-kube-descheduler-operator
```

2. **deschedulingIntervalSeconds** フィールドを必要な値に更新します。

```
apiVersion: operator.openshift.io/v1beta1
kind: KubeDescheduler
metadata:
  name: cluster
  namespace: openshift-kube-descheduler-operator
spec:
  deschedulingIntervalSeconds: 3600 (1)
...
```

(1) Descheduler の実行間隔を秒単位で設定します。このフィールドの値 **0** は Descheduler を一度実行し、終了します。

3. 変更を適用するためにファイルを保存します。

3.11.6. Descheduler のアンインストール

Descheduler インスタンスを削除し、Kube Descheduler Operator をアンインストールして Descheduler をクラスターから削除できます。この手順では、**KubeDescheduler** CRD および **openshift-kube-descheduler-operator** namespace もクリーンアップします。

前提条件

- クラスター管理者の権限。
- OpenShift Container Platform Web コンソールにアクセスします。

手順

1. OpenShift Container Platform Web コンソールにログインします。
2. Descheduler インスタンスを削除します。
   a. **Operators → Installed Operators** ページから、**Kube Descheduler Operator** をクリックします。
   b. **Kube Descheduler** タブを選択します。
   c. **cluster** クラスターの横にある Options メニュー ⋮ をクリックし、**Delete KubeDescheduler** を選択します。

d. 確認ダイアログで **Delete** をクリックします。

3. Kube Descheduler Operator をアンインストールします。

a. **Operators → Installed Operators** に移動します。

b. **Kube Descheduler Operator** エントリーの横にある Options メニュー  をクリックし、**Uninstall Operator** を選択します。

c. 確認ダイアログで、**Uninstall** をクリックします。

4. **openshift-kube-descheduler-operator** namespace を削除します。

a. **Administration → Namespaces** に移動します。

b. **openshift-kube-descheduler-operator** をフィルターボックスに入力します。

c. **openshift-kube-descheduler-operator** エントリーの横にある Options メニュー  をクリックし、**Delete Namespace** を選択します。

d. 確認ダイアログで **openshift-kube-descheduler-operator** を入力し、**Delete** をクリックします。

5. **KubeDescheduler** CRD を削除します。

a. **Administration → Custom Resource Definitions**に移動します。

b. **KubeDescheduler** をフィルターボックスに入力します。

c. **KubeDescheduler** エントリーの横にある Options メニュー  をクリックし、**Delete CustomResourceDefinition** を選択します。

d. 確認ダイアログで **Delete** をクリックします。

第4章 ジョブと DEAMONSET の使用

4.1. デーモンセットによるノード上でのバックグラウンドタスクの自動的な実行

管理者は、デーモンセットを作成して OpenShift Container Platform クラスター内の特定の、またはすべてのノードで Pod のレプリカを実行するために使用できます。

デーモンセットは、すべて (または一部) のノードで Pod のコピーが確実に実行されるようにします。ノードがクラスターに追加されると、Pod がクラスターに追加されます。ノードがクラスターから削除されると、Pod はガベージコレクションによって削除されます。デーモンセットを削除すると、デーモンセットによって作成された Pod がクリーンアップされます。

デーモンセットを使用して共有ストレージを作成し、クラスター内のすべてのノードでロギング Pod を実行するか、またはすべてのノードでモニターエージェントをデプロイできます。

セキュリティー上の理由から、 クラスター管理者のみがデーモンセットを作成できます。

デーモンセットについての詳細は、Kubernetes ドキュメント を参照してください。

重要

デーモンセットのスケジューリングにはプロジェクトのデフォルトノードセレクターとの互換性がありません。これを無効にしない場合、デーモンセットはデフォルトのノードセレクターとのマージによって制限されます。これにより、マージされたノードセレクターで選択解除されたノードで Pod が頻繁に再作成されるようになり、クラスターに不要な負荷が加わります。

4.1.1. デフォルトスケジューラーによるスケジュール

デーモンセットは、適格なすべてのノードで Pod のコピーが確実に実行されるようにします。通常は、Pod が実行されるノードは Kubernetes のスケジューラーが選択します。ただし、これまでデーモンセット Pod はデーモンセットコントローラーが作成し、スケジュールしていました。その結果、以下のような問題が生じています。

- Pod の動作に一貫性がない。スケジューリングを待機している通常の Pod は、作成されると Pending 状態になりますが、デーモンセット Pod は作成されても **Pending** 状態になりません。これによりユーザーに混乱が生じます。
- Pod のプリエンプションがデフォルトのスケジューラーで処理される。プリエンプションが有効にされると、デーモンセットコントローラーは Pod の優先順位とプリエンプションを考慮することなくスケジューリングの決定を行います。

ScheduleDaemonSetPods 機能は、OpenShift Container Platform でデフォルトで有効にされます。これにより、**spec.nodeName** の条件 (term) ではなく **NodeAffinity** の条件 (term) をデーモンセット Pod に追加することで、デーモンセットコントローラーではなくデフォルトのスケジューラーを使ってデーモンセットをスケジュールすることができます。その後、デフォルトのスケジューラーは、Pod をターゲットホストにバインドさせるために使用されます。デーモンセット Pod のノードアフィニティーがすでに存在する場合、これは置き換えられます。デーモンセットコントローラーは、デーモンセット Pod を作成または変更する場合にのみこれらの操作を実行し、デーモンセットの **spec.template** は一切変更されません。

```
nodeAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
```

```
nodeSelectorTerms:
- matchFields:
  - key: metadata.name
    operator: In
    values:
    - target-host-name
```

さらに、**node.kubernetes.io/unschedulable:NoSchedule** の容認がデーモンセット Pod に自動的に追加されます。デフォルトのスケジューラーは、デーモンセット Pod をスケジュールする際に、スケジュールできないノードを無視します。

4.1.2. デーモンセットの作成

デーモンセットの作成時に、**nodeSelector** フィールドは、デーモンセットがレプリカをデプロイする必要のあるノードを指定するために使用されます。

前提条件

- デーモンセットの使用を開始する前に、namespace のアノテーション **openshift.io/node-selector** を空の文字列に設定することで、namespace のプロジェクトスコープのデフォルトのノードセレクターを無効にします。

```
$ oc patch namespace myproject -p \
    '{"metadata": {"annotations": {"openshift.io/node-selector": ""}}}'
```

- 新規プロジェクトを作成している場合は、デフォルトのノードセレクターを上書きします。

```
`oc adm new-project <name> --node-selector=""`.
```

手順

デーモンセットを作成するには、以下を実行します。

1. デーモンセット yaml ファイルを定義します。

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: hello-daemonset
spec:
  selector:
     matchLabels:
       name: hello-daemonset (1)
  template:
    metadata:
      labels:
        name: hello-daemonset (2)
    spec:
      nodeSelector: (3)
        role: worker
      containers:
      - image: openshift/hello-openshift
        imagePullPolicy: Always
        name: registry
```

```
      ports:
      - containerPort: 80
        protocol: TCP
      resources: {}
      terminationMessagePath: /dev/termination-log
    serviceAccount: default
    terminationGracePeriodSeconds: 10
```

1. デーモンセットに属する Pod を判別するラベルセレクターです。
2. Pod テンプレートのラベルセレクターです。上記のラベルセレクターに一致している必要があります。
3. Pod レプリカをデプロイする必要があるノードを判別するノードセレクターです。一致するラベルがこのノードに存在する必要があります。

2. デーモンセットオブジェクトを作成します。

```
$ oc create -f daemonset.yaml
```

3. Pod が作成されていることを確認し、各 Pod に Pod レプリカがあることを確認するには、以下を実行します。

   a. daemonset Pod を検索します。

   ```
   $ oc get pods
   ```

   出力例

   ```
   hello-daemonset-cx6md   1/1       Running   0          2m
   hello-daemonset-e3md9   1/1       Running   0          2m
   ```

   b. Pod がノードに配置されていることを確認するために Pod を表示します。

   ```
   $ oc describe pod/hello-daemonset-cx6md|grep Node
   ```

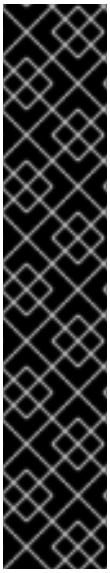
   出力例

   ```
   Node:        openshift-node01.hostname.com/10.14.20.134
   ```

   ```
   $ oc describe pod/hello-daemonset-e3md9|grep Node
   ```

   出力例

   ```
   Node:        openshift-node02.hostname.com/10.14.20.137
   ```



重要

- デーモンセット Pod テンプレートを更新しても、既存の Pod レプリカには影響はありません。
- デーモンセットを削除してから、異なるテンプレートと同じラベルセレクターを使用して新規のデーモンセットを作成する場合に、既存の Pod レプリカについてラベルが一致していると認識するため、既存の Pod レプリカは更新されず、Pod テンプレートで一致しない場合でも新しいレプリカが作成されます。
- ノードのラベルを変更する場合には、デーモンセットは新しいラベルと一致するノードに Pod を追加し、新しいラベルと一致しないノードから Pod を削除します。

デーモンセットを更新するには、古いレプリカまたはノードを削除して新規の Pod レプリカの作成を強制的に実行します。

4.2. ジョブの使用による POD でのタスクの実行

job は、OpenShift Container Platform クラスターのタスクを実行します。

ジョブは、タスクの全体的な進捗状況を追跡し、進行中、完了、および失敗した各 Pod の情報を使ってその状態を更新します。ジョブを削除するとそのジョブによって作成された Pod のレプリカがクリーンアップされます。ジョブは Kubernetes API の一部で、他のオブジェクトタイプ同様に **oc** コマンドで管理できます。

ジョブ仕様のサンプル

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  parallelism: 1    (1)
  completions: 1    (2)
  activeDeadlineSeconds: 1800 (3)
  backoffLimit: 6   (4)
  template:         (5)
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: OnFailure    (6)
```

(1) ジョブの Pod レプリカは並行して実行される必要があります。

(2) ジョブの完了をマークするには、Pod の正常な完了が必要です。

(3) ジョブを実行できる最長期間。

(4) ジョブの再試行回数。

5 コントローラーが作成する Pod のテンプレート。

6 Pod の再起動ポリシー。

ジョブについての詳細は、Kubernetes のドキュメント を参照してください。

4.2.1. ジョブと Cron ジョブについて

ジョブは、タスクの全体的な進捗状況を追跡し、進行中、完了、および失敗した各 Pod の情報を使ってその状態を更新します。ジョブを削除するとそのジョブによって作成された Pod がクリーンアップされます。ジョブは Kubernetes API の一部で、他のオブジェクトタイプ同様に **oc** コマンドで管理できます。

OpenShift Container Platform で一度だけ実行するオブジェクトを作成できるリソースタイプは 2 種類あります。

ジョブ

定期的なジョブは、タスクを作成しジョブが完了したことを確認する、一度だけ実行するオブジェクトです。

ジョブとして実行するには、主に以下のタスクタイプを使用できます。

- 非並列ジョブ:
  - Pod が失敗しない限り、単一の Pod のみを起動するジョブ。
  - このジョブは、Pod が正常に終了するとすぐに完了します。
- 固定の完了数が指定された並列ジョブ
  - 複数の Pod を起動するジョブ。
  - ジョブはタスク全体を表し、**1** から **completions** 値までの範囲内のそれぞれの値に対して 1 つの正常な Pod がある場合に完了します。
- ワークキューを含む並列ジョブ:
  - 指定された Pod に複数の並列ワーカープロセスを持つジョブ。
  - OpenShift Container Platform は Pod を調整し、それぞれの機能を判別するか、または外部キューサービスを使用します。
  - 各 Pod はそれぞれ、すべてのピア Pod が完了しているかどうかや、ジョブ全体が実行済みであることを判別することができます。
  - ジョブからの Pod が正常な状態で終了すると、新規 Pod は作成されません。
  - 1 つ以上の Pod が正常な状態で終了し、すべての Pod が終了している場合、ジョブが正常に完了します。
  - Pod が正常な状態で終了した場合、それ以外の Pod がこのタスクについて機能したり、または出力を書き込むことはありません。Pod はすべて終了プロセスにあるはずです。

各種のジョブを使用する方法についての詳細は、Kubernetes ドキュメントの Job Patterns を参照してください。

Cron ジョブ

ジョブは、Cron ジョブを使って複数回実行するようにスケジュールすることが可能です。

cron ジョブ は、ユーザーがジョブの実行方法を指定することを可能にすることで、定期的なジョブを積み重ねます。Cron ジョブは Kubernetes API の一部であり、他のオブジェクトタイプと同様に **oc** コマンドで管理できます。

Cron ジョブは、バックアップの実行やメールの送信など周期的な繰り返しのタスクを作成する際に役立ちます。また、低アクティビティー期間にジョブをスケジュールする場合など、特定の時間に個別のタスクをスケジュールすることも可能です。cron ジョブは、cronjob コントローラーを実行するコントロールプレーンノードに設定されたタイムゾーンに基づいて **Job** オブジェクトを作成します。

警告

Cron ジョブはスケジュールの実行時間ごとに約 1 回ずつ **Job** オブジェクトを作成しますが、ジョブの作成に失敗したり、2 つのジョブが作成される場合があります。そのためジョブはべき等である必要があり、履歴制限を設定する必要があります。

4.2.1.1. ジョブの作成方法

どちらのリソースタイプにも、以下の主要な要素から設定されるジョブ設定が必要です。

- OpenShift Container Platform が作成する Pod を記述している Pod テンプレート。
- **parallelism** パラメーター。ジョブの実行に使用する、同時に実行される Pod の数を指定します。
  - 非並列ジョブの場合は、未設定のままにします。未設定の場合は、デフォルトの **1** に設定されます。
- **completions** パラメーター。ジョブを完了するために必要な、正常に完了した Pod の数を指定します。
  - 非並列ジョブの場合は、未設定のままにします。未設定の場合は、デフォルトの **1** に設定されます。
  - 固定の完了数を持つ並列ジョブの場合は、値を指定します。
  - ワークキューのある並列ジョブでは、未設定のままにします。未設定の場合、デフォルトは **parallelism** 値に設定されます。

4.2.1.2. ジョブの最長期間を設定する方法

ジョブの定義時に、**activeDeadlineSeconds** フィールドを設定して最長期間を定義できます。これは秒単位で指定され、デフォルトでは設定されません。設定されていない場合は、実施される最長期間はありません。

最長期間は、最初の Pod がスケジュールされた時点から計算され、ジョブが有効である期間を定義します。これは実行の全体の時間を追跡します。指定されたタイムアウトに達すると、OpenShift Container Platform がジョブを終了します。

4.2.1.3. 失敗した Pod のためのジョブのバックオフポリシーを設定する方法

ジョブは、設定の論理的なエラーなどの理由により再試行の設定回数を超えた後に失敗とみなされる場合があります。ジョブに関連付けられた失敗した Pod は 6 分を上限として指数関数的バックオフ遅延値 (**10s**、**20s**、**40s** ...) に基づいて再作成されます。この制限は、コントローラーのチェック間で失敗した Pod が新たに生じない場合に再設定されます。

ジョブの再試行回数を設定するには **spec.backoffLimit** パラメーターを使用します。

4.2.1.4. アーティファクトを削除するように Cron ジョブを設定する方法

Cron ジョブはジョブや Pod などのアーティファクトリーソースをそのままにすることがあります。ユーザーは履歴制限を設定して古いジョブとそれらの Pod が適切に消去されるようにすることが重要です。これに対応する 2 つのフィールドが Cron ジョブ仕様にあります。

- **.spec.successfulJobsHistoryLimit**.保持する成功した終了済みジョブの数 (デフォルトは 3 に設定)。
- **.spec.successfulJobsHistoryLimit**。保持する失敗した終了済みジョブの数 (デフォルトは 1 に設定)。

ヒント

- 必要なくなった Cron ジョブを削除します。

```
$ oc delete cronjob/<cron_job_name>
```

  これを実行することで、不要なアーティファクトの生成を防げます。

- **spec.suspend** を true に設定することで、その後の実行を中断することができます。その後のすべての実行は、**false** に再設定するまで中断されます。

4.2.1.5. 既知の制限

ジョブ仕様の再起動ポリシーは **Pod** にのみ適用され、**ジョブコントローラー** には適用されません。ただし、ジョブコントローラーはジョブを完了まで再試行するようハードコーディングされます。

そのため **restartPolicy: Never** または **--restart=Never** により、**restartPolicy: OnFailure** または **--restart=OnFailure** と同じ動作が実行されます。つまり、ジョブが失敗すると、成功するまで (または手動で破棄されるまで) 自動で再起動します。このポリシーは再起動するサブシステムのみを設定します。

Never ポリシーでは、**ジョブコントローラー** が再起動を実行します。それぞれの再試行時に、ジョブコントローラーはジョブステータスの失敗数を増分し、新規 Pod を作成します。これは、それぞれの試行が失敗するたびに Pod の数が増えることを意味します。

OnFailure ポリシーでは、**kubelet** が再起動を実行します。それぞれの試行によりジョブステータスでの失敗数が増分する訳ではありません。さらに、kubelet は同じノードで Pod の起動に失敗したジョブを再試行します。

4.2.2. ジョブの作成

ジョブオブジェクトを作成して OpenShift Container Platform にジョブを作成します。

手順

ジョブを作成するには、以下を実行します。

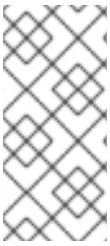
1. 以下のような YAML ファイルを作成します。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  parallelism: 1    (1)
  completions: 1    (2)
  activeDeadlineSeconds: 1800 (3)
  backoffLimit: 6   (4)
  template:         (5)
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: OnFailure    (6)
```

1. オプションで、ジョブが並列で実行される Pod のレプリカ数を指定します。デフォルトは **1** に設定されます。
   - 非並列ジョブの場合は、未設定のままにします。未設定の場合は、デフォルトの **1** に設定されます。
2. オプションで、ジョブを完了 (completed) としてマークするために必要な Pod の正常な完了数を指定します。
   - 非並列ジョブの場合は、未設定のままにします。未設定の場合は、デフォルトの **1** に設定されます。
   - 固定の完了数を持つ並列ジョブの場合、完了の数を指定します。
   - ワークキューのある並列ジョブでは、未設定のままにします。未設定の場合、デフォルトは **parallelism** 値に設定されます。
3. オプションで、ジョブを実行できる最長期間を指定します。
4. オプションで、ジョブの再試行回数を指定します。このフィールドは、デフォルトでは 6 に設定されています。
5. コントローラーが作成する Pod のテンプレートを指定します。
6. Pod の再起動ポリシーを指定します。
   - **Never**.ジョブを再起動しません。
   - **OnFailure**.ジョブが失敗した場合にのみ再起動します。
   - **Always**ジョブを常に再起動します。
     OpenShift Container Platform が失敗したコンテナーについて再起動ポリシーを使用する方法の詳細は、Kubernetes ドキュメントの State の例 を参照してください。

2. ジョブを作成します。

```
$ oc create -f <file-name>.yaml
```



注記

oc create job を使用して単一コマンドからジョブを作成し、起動することもできます。以下のコマンドは直前の例に指定されている同じジョブを作成し、これを起動します。

```
$ oc create job pi --image=perl -- perl -Mbignum=bpi -wle 'print bpi(2000)'
```

4.2.3. cron ジョブの作成

ジョブオブジェクトを作成して OpenShift Container Platform に cron ジョブを作成します。

手順

cron ジョブを作成するには、以下を実行します。

1. 以下のような YAML ファイルを作成します。

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: pi
spec:
  schedule: "*/1 * * * *" 1
  concurrencyPolicy: "Replace" 2
  startingDeadlineSeconds: 200 3
  suspend: true 4
  successfulJobsHistoryLimit: 3 5
  failedJobsHistoryLimit: 1 6
  jobTemplate: 7
    spec:
      template:
        metadata:
          labels: 8
            parent: "cronjobpi"
        spec:
          containers:
          - name: pi
            image: perl
            command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
          restartPolicy: OnFailure 9
```

1 1 cron 形式 で指定されたジョブのスケジュール。この例では、ジョブは毎分実行されます。

2 2 オプションの同時実行ポリシー。cron ジョブ内での同時実行ジョブを処理する方法を指定します。以下の同時実行ポリシーの 1 つのみを指定できます。これが指定されない場合、同時実行を許可するようにデフォルト設定されます。

- **Allow**: Cron ジョブを同時に実行できます。

- **Forbid**: 同時実行を禁止し、直前の実行が終了していない場合は次の実行を省略します。
- **Replace**: 同時に実行されているジョブを取り消し、これを新規ジョブに置き換えます。

3 3 ジョブを開始するためのオプションの期限 (秒単位)(何らかの理由によりスケジュールされた時間が経過する場合)。ジョブの実行が行われない場合、ジョブの失敗としてカウントされます。これが指定されない場合は期間が設定されません。

4 4 Cron ジョブの停止を許可するオプションのフラグ。これが **true** に設定されている場合、後続のすべての実行が停止されます。

5 5 保持する成功した終了済みジョブの数 (デフォルトは 3 に設定)。

6 6 保持する失敗した終了済みジョブの数 (デフォルトは 1 に設定)。

7 ジョブテンプレート。これはジョブの例と同様です。

8 この Cron ジョブで生成されるジョブのラベルを設定します。

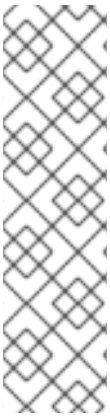
9 Pod の再起動ポリシー。ジョブコントローラーには適用されません。

注記

.spec.successfulJobsHistoryLimit と **.spec.failedJobsHistoryLimit** のフィールドはオプションです。これらのフィールドでは、完了したジョブと失敗したジョブのそれぞれを保存する数を指定します。デフォルトで、これらのジョブの保存数はそれぞれ **3** と **1** に設定されます。制限に **0** を設定すると、終了後に対応する種類のジョブのいずれも保持しません。

2. cron ジョブを作成します。

```
$ oc create -f <file-name>.yaml
```



注記

oc create cronjob を使用して単一コマンドから cron ジョブを作成し、起動することもできます。以下のコマンドは直前の例で指定されている同じ cron ジョブを作成し、これを起動します。

```
$ oc create cronjob pi --image=perl --schedule='*/1 * * * *' -- perl -Mbignum=bpi -wle
'print bpi(2000)'
```

oc create cronjob で、**--schedule** オプションは cron 形式 のスケジュールを受け入れます。

第5章 ノードの使用

5.1. OPENSHIFT CONTAINER PLATFORM クラスター内のノードの閲覧と一覧表示

クラスターのすべてのノードを一覧表示し、ステータスや経過時間、メモリー使用量などの情報およびノードについての詳細を取得できます。

ノード管理の操作を実行すると、CLI は実際のノードホストの表現であるノードオブジェクトと対話します。マスターはノードオブジェクトの情報を使ってヘルスチェックでノードを検証します。

5.1.1. クラスター内のすべてのノードの一覧表示について

クラスター内のノードに関する詳細な情報を取得できます。

- 以下のコマンドは、すべてのノードを一覧表示します。

```
$ oc get nodes
```

以下の例は、正常なノードを持つクラスターです。

```
$ oc get nodes
```

出力例

```
NAME                STATUS    ROLES     AGE       VERSION
master.example.com  Ready     master    7h        v1.20.0
node1.example.com   Ready     worker    7h        v1.20.0
node2.example.com   Ready     worker    7h        v1.20.0
```

以下の例は、正常でないノードが1つ含まれるクラスターです。

```
$ oc get nodes
```

出力例

```
NAME                STATUS                     ROLES     AGE       VERSION
master.example.com  Ready                      master    7h        v1.20.0
node1.example.com   NotReady,SchedulingDisabled worker    7h        v1.20.0
node2.example.com   Ready                      worker    7h        v1.20.0
```

NotReady ステータスをトリガーする条件については、本セクションの後半で説明します。

- **-o wide** オプションは、ノードについての追加情報を提供します。

```
$ oc get nodes -o wide
```

出力例

```
NAME                STATUS   ROLES    AGE    VERSION        INTERNAL-IP    EXTERNAL-IP
OS-IMAGE                                KERNEL-VERSION                CONTAINER-
```

```
RUNTIME
master.example.com  Ready    master   171m   v1.20.0+39c0afe   10.0.129.108   <none>
Red Hat Enterprise Linux CoreOS 48.83.202103210901-0 (Ootpa)   4.18.0-
240.15.1.el8_3.x86_64   cri-o://1.21.0-30.rhaos4.8.gitf2f339d.el8-dev
node1.example.com   Ready    worker   72m    v1.20.0+39c0afe   10.0.129.222   <none>
Red Hat Enterprise Linux CoreOS 48.83.202103210901-0 (Ootpa)   4.18.0-
240.15.1.el8_3.x86_64   cri-o://1.21.0-30.rhaos4.8.gitf2f339d.el8-dev
node2.example.com   Ready    worker   164m   v1.20.0+39c0afe   10.0.142.150   <none>
Red Hat Enterprise Linux CoreOS 48.83.202103210901-0 (Ootpa)   4.18.0-
240.15.1.el8_3.x86_64   cri-o://1.21.0-30.rhaos4.8.gitf2f339d.el8-dev
```

- 以下のコマンドは、単一のノードに関する情報を一覧表示します。

```
$ oc get node <node>
```

以下に例を示します。

```
$ oc get node node1.example.com
```

出力例

```
NAME                STATUS    ROLES     AGE       VERSION
node1.example.com   Ready     worker    7h        v1.20.0
```

- 以下のコマンドを実行すると、現在の状態の理由を含む、特定ノードについての詳細情報を取得できます。

```
$ oc describe node <node>
```

以下に例を示します。

```
$ oc describe node node1.example.com
```

出力例

```
Name:               node1.example.com 1
Roles:              worker 2
Labels:             beta.kubernetes.io/arch=amd64 3
                    beta.kubernetes.io/instance-type=m4.large
                    beta.kubernetes.io/os=linux
                    failure-domain.beta.kubernetes.io/region=us-east-2
                    failure-domain.beta.kubernetes.io/zone=us-east-2a
                    kubernetes.io/hostname=ip-10-0-140-16
                    node-role.kubernetes.io/worker=
Annotations:        cluster.k8s.io/machine: openshift-machine-api/ahardin-worker-us-east-2a-
q5dzc 4
                    machineconfiguration.openshift.io/currentConfig: worker-
309c228e8b3a92e2235edd544c62fea8
                    machineconfiguration.openshift.io/desiredConfig: worker-
309c228e8b3a92e2235edd544c62fea8
                    machineconfiguration.openshift.io/state: Done
                    volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp:  Wed, 13 Feb 2019 11:05:57 -0500
```

```
Taints:             <none>  5
Unschedulable:      false
Conditions:                 6
  Type             Status  LastHeartbeatTime                 LastTransitionTime                Reason
Message
  ----             ------  -----------------                 ------------------                ------
  OutOfDisk        False   Wed, 13 Feb 2019 15:09:42 -0500   Wed, 13 Feb 2019 11:05:57 -
0500   KubeletHasSufficientDisk     kubelet has sufficient disk space available
  MemoryPressure   False   Wed, 13 Feb 2019 15:09:42 -0500   Wed, 13 Feb 2019 11:05:57
-0500   KubeletHasSufficientMemory   kubelet has sufficient memory available
  DiskPressure     False   Wed, 13 Feb 2019 15:09:42 -0500   Wed, 13 Feb 2019 11:05:57 -
0500   KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure      False   Wed, 13 Feb 2019 15:09:42 -0500   Wed, 13 Feb 2019 11:05:57 -
0500   KubeletHasSufficientPID      kubelet has sufficient PID available
  Ready            True    Wed, 13 Feb 2019 15:09:42 -0500   Wed, 13 Feb 2019 11:07:09 -0500
KubeletReady                 kubelet is posting ready status
Addresses:                  7
  InternalIP:   10.0.140.16
  InternalDNS:  ip-10-0-140-16.us-east-2.compute.internal
  Hostname:     ip-10-0-140-16.us-east-2.compute.internal
Capacity:                   8
 attachable-volumes-aws-ebs:  39
 cpu:                         2
 hugepages-1Gi:               0
 hugepages-2Mi:               0
 memory:                      8172516Ki
 pods:                        250
Allocatable:
 attachable-volumes-aws-ebs:  39
 cpu:                         1500m
 hugepages-1Gi:               0
 hugepages-2Mi:               0
 memory:                      7558116Ki
 pods:                        250
System Info:                9
 Machine ID:                             63787c9534c24fde9a0cde35c13f1f66
 System UUID:                            EC22BF97-A006-4A58-6AF8-0A38DEEA122A
 Boot ID:                                f24ad37d-2594-46b4-8830-7f7555918325
 Kernel Version:                         3.10.0-957.5.1.el7.x86_64
 OS Image:                               Red Hat Enterprise Linux CoreOS 410.8.20190520.0 (Ootpa)
 Operating System:                       linux
 Architecture:                           amd64
 Container Runtime Version:              cri-o://1.16.0-0.6.dev.rhaos4.3.git9ad059b.el8-rc2
 Kubelet Version:                        v1.20.0
 Kube-Proxy Version:                     v1.20.0
PodCIDR:                                 10.128.4.0/24
ProviderID:                              aws:///us-east-2a/i-04e87b31dc6b3e171
Non-terminated Pods:                     (13 in total)  10
  Namespace                              Name                                  CPU Requests  CPU Limits
Memory Requests  Memory Limits
  ---------                              ----                                  ------------  ----------  ---------------  -------
------
  openshift-cluster-node-tuning-operator  tuned-hdl5q                          0 (0%)        0 (0%)      0
(0%)          0 (0%)
  openshift-dns                          dns-default-l69zr                     0 (0%)        0 (0%)      0 (0%)
```

```
0 (0%)
  openshift-image-registry              node-ca-9hmcg                        0 (0%)       0 (0%)      0
(0%)          0 (0%)
  openshift-ingress                     router-default-76455c45c-c5ptv       0 (0%)       0 (0%)      0
(0%)          0 (0%)
  openshift-machine-config-operator     machine-config-daemon-cvqw9          20m (1%)     0
(0%)      50Mi (0%)       0 (0%)
  openshift-marketplace                 community-operators-f67fh            0 (0%)       0 (0%)
0 (0%)          0 (0%)
  openshift-monitoring                  alertmanager-main-0                  50m (3%)     50m (3%)
210Mi (2%)      10Mi (0%)
  openshift-monitoring                  grafana-78765ddcc7-hnjmm             100m (6%)    200m
(13%)  100Mi (1%)      200Mi (2%)
  openshift-monitoring                  node-exporter-l7q8d                  10m (0%)     20m (1%)
20Mi (0%)       40Mi (0%)
  openshift-monitoring                  prometheus-adapter-75d769c874-hvb85  0 (0%)       0
(0%)      0 (0%)          0 (0%)
  openshift-multus                      multus-kw8w5                         0 (0%)       0 (0%)      0 (0%)
0 (0%)
  openshift-sdn                         ovs-t4dsn                            100m (6%)    0 (0%)      300Mi
(4%)      0 (0%)
  openshift-sdn                         sdn-g79hg                            100m (6%)    0 (0%)      200Mi
(2%)      0 (0%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource                    Requests     Limits
  --------                    --------     ------
  cpu                         380m (25%)   270m (18%)
  memory                      880Mi (11%)  250Mi (3%)
  attachable-volumes-aws-ebs  0            0
Events:                                                                                                    (11)
  Type     Reason                   Age                From                        Message
  ----     ------                   ----               ----                        -------
  Normal   NodeHasSufficientPID     6d (x5 over 6d)    kubelet, m01.example.com    Node
m01.example.com status is now: NodeHasSufficientPID
  Normal   NodeAllocatableEnforced  6d                 kubelet, m01.example.com    Updated Node
Allocatable limit across pods
  Normal   NodeHasSufficientMemory  6d (x6 over 6d)    kubelet, m01.example.com    Node
m01.example.com status is now: NodeHasSufficientMemory
  Normal   NodeHasNoDiskPressure    6d (x6 over 6d)    kubelet, m01.example.com    Node
m01.example.com status is now: NodeHasNoDiskPressure
  Normal   NodeHasSufficientDisk    6d (x6 over 6d)    kubelet, m01.example.com    Node
m01.example.com status is now: NodeHasSufficientDisk
  Normal   NodeHasSufficientPID     6d                 kubelet, m01.example.com    Node
m01.example.com status is now: NodeHasSufficientPID
  Normal   Starting                 6d                 kubelet, m01.example.com    Starting kubelet.
 ...
```

1. ノードの名前。
2. ノードのロール (**master** または **worker** のいずれか)。
3. ノードに適用されたラベル。
4. ノードに適用されるアノテーション。

5 ノードに適用されたテイント。

6 ノードの状態およびステータス。**conditions** スタンザは、**Ready**、**PIDPressure**、**PIDPressure**、**MemoryPressure**、 **DiskPressure** および **OutOfDisk** ステータスを一覧表示します。これらの状態については、本セクションの後半で説明します。

7 ノードの IP アドレスとホスト名。

8 Pod のリソースと割り当て可能なリソース。

9 ノードホストについての情報。

10 ノードの Pod。

11 ノードが報告したイベント。

ノードについての情報の中でも、とりわけ以下のノードの状態がこのセクションで説明されるコマンドの出力に表示されます。

表5.1 ノードの状態

状態	説明
Ready	**true** の場合、ノードは正常であり、Pod を受け入れることのできる準備状態にあります。**false** の場合、ノードは正常ではなく、Pod を受け入れません。**unknown** の場合、ノードコントローラーは **node-monitor-grace-period** (デフォルトは 40 秒) の間にハートビートをノードから受信しませんでした。
DiskPressure	**true** の場合、ディスク容量は低くなります。
MemoryPressure	**true** の場合、ノードのメモリーは低くなります。
PIDPressure	**true** の場合、ノードのプロセスが多すぎます。
OutOfDisk	**true** の場合、ノードには新しい Pod を追加するためのノード上の空きスペースが十分にありません。
NetworkUnavailable	**true** の場合、ノードのネットワークは正しく設定されていません。
NotReady	**true** の場合、コンテナーのランタイムやネットワークなど基本のコンポーネントのいずれかに問題が発生しているか、またはそれらがまだ設定されていません。
SchedulingDisabled	ノードに配置するように Pod をスケジュールすることができません。

5.1.2. クラスターでのノード上の Pod の一覧表示

特定のノード上のすべての Pod を一覧表示できます。

手順

- 1 つ以上のノードにすべてまたは選択した Pod を一覧表示するには、以下を実行します。

```
$ oc describe node <node1> <node2>
```

以下に例を示します。

```
$ oc describe node ip-10-0-128-218.ec2.internal
```

- 選択したノードのすべてまたは選択した Pod を一覧表示するには、以下を実行します。

```
$ oc describe --selector=<node_selector>
```

```
$ oc describe node  --selector=kubernetes.io/os
```

または、以下を実行します。

```
$ oc describe -l=<pod_selector>
```

```
$ oc describe node -l node-role.kubernetes.io/worker
```

- 終了した Pod を含む、特定のノード上のすべての Pod を一覧表示するには、以下を実行します。

```
$ oc get pod --all-namespaces --field-selector=spec.nodeName=<nodename>
```

5.1.3. ノードのメモリーと CPU 使用統計の表示

コンテナーのランタイム環境を提供する、ノードについての使用状況の統計を表示できます。これらの使用状況の統計には CPU、メモリー、およびストレージの消費量が含まれます。

前提条件

- 使用状況の統計を表示するには、**cluster-reader** パーミッションがなければなりません。
- 使用状況の統計を表示するには、メトリクスをインストールしている必要があります。

手順

- 使用状況の統計を表示するには、以下を実行します。

```
$ oc adm top nodes
```

出力例

```
NAME                                CPU(cores)   CPU%      MEMORY(bytes)   MEMORY%
ip-10-0-12-143.ec2.compute.internal   1503m        100%      4533Mi          61%
ip-10-0-132-16.ec2.compute.internal   76m          5%        1391Mi          18%
ip-10-0-140-137.ec2.compute.internal  398m         26%       2473Mi          33%
ip-10-0-142-44.ec2.compute.internal   656m         43%       6119Mi          82%
```

```
ip-10-0-146-165.ec2.compute.internal   188m         12%       3367Mi          45%
ip-10-0-19-62.ec2.compute.internal     896m         59%       5754Mi          77%
ip-10-0-44-193.ec2.compute.internal    632m         42%       5349Mi          72%
```

- ラベルの付いたノードの使用状況の統計を表示するには、以下を実行します。

```
$ oc adm top node --selector=''
```

フィルターに使用するセレクター (ラベルクエリー) を選択する必要があります。**=**、**==**、および **!=** をサポートします。

5.2. ノードの使用

管理者として、クラスターの効率をさらに上げる多数のタスクを実行することができます。

5.2.1. ノード上の Pod を退避させる方法

Pod を退避させると、所定のノードからすべての Pod または選択した Pod を移行できます。

退避させることができるのは、レプリケーションコントローラーが管理している Pod のみです。レプリケーションコントローラーは、他のノードに新しい Pod を作成し、指定されたノードから既存の Pod を削除します。

ベア Pod、つまりレプリケーションコントローラーが管理していない Pod はデフォルトで影響を受けません。Pod セレクターを指定すると Pod のサブセットを退避できます。Pod セレクターはラベルに基づくので、指定したラベルを持つすべての Pod を退避できます。

手順

1. Pod の退避を実行する前に、ノードをスケジュール対象外としてマークします。
   a. ノードにスケジュール対象外 (unschedulable) のマークを付けます。

```
$ oc adm cordon <node1>
```

出力例

```
node/<node1> cordoned
```

   b. ノードのステータスが **Ready,SchedulingDisabled** であることを確認します。

```
$ oc get node <node1>
```

出力例

```
NAME       STATUS                     ROLES    AGE      VERSION
<node1>    Ready,SchedulingDisabled   worker   1d       v1.24.0
```

2. 以下の方法のいずれかを使用して Pod を退避します。
   - 1 つ以上のノードで、すべてまたは選択した Pod を退避します。

```
$ oc adm drain <node1> <node2> [--pod-selector=<pod_selector>]
```

-

- **--force** オプションを使用してベア Pod の削除を強制的に実行します。**true** に設定されると、Pod がレプリケーションコントローラー、レプリカセット、ジョブ、デーモンセット、またはステートフルセットで管理されていない場合でも削除が続行されます。

```
$ oc adm drain <node1> <node2> --force=true
```

- **--grace-period** を使用して、各 Pod を正常に終了するための期間 (秒単位) を設定します。負の値の場合には、Pod に指定されるデフォルト値が使用されます。

```
$ oc adm drain <node1> <node2> --grace-period=-1
```

- **true** に設定された **--ignore-daemonsets** フラグを使用してデーモンセットが管理する Pod を無視します。

```
$ oc adm drain <node1> <node2> --ignore-daemonsets=true
```

- **--timeout** を使用して、中止する前の待機期間を設定します。値 **0** は無限の時間を設定します。

```
$ oc adm drain <node1> <node2> --timeout=5s
```

- **--delete-emptydir-data** フラグを **true** に設定して、**emptyDir** ボリュームを使用する Pod がある場合にも Pod を削除します。ローカルデータはノードがドレイン (解放) される場合に削除されます。

```
$ oc adm drain <node1> <node2> --delete-emptydir-data=true
```

- **true** に設定された **--dry-run** オプションを使用して、実際に退避を実行せずに移行するオブジェクトを一覧表示します。

```
$ oc adm drain <node1> <node2>  --dry-run=true
```

  特定のノード名 (例: **<node1> <node2>**) を指定する代わりに、**--selector=<node_selector>** オプションを使用し、選択したノードで Pod を退避することができます。

3. 完了したら、ノードにスケジュール対象のマークを付けます。

```
$ oc adm uncordon <node1>
```

5.2.2. ノードでラベルを更新する方法について

ノード上の任意のラベルを更新できます。

ノードラベルは、ノードがマシンによってバックアップされている場合でも、ノードが削除されると永続しません。

注記

MachineSet への変更は、マシンセットが所有する既存のマシンには適用されません。たとえば、編集されたか、または既存の **MachineSet** に追加されたラベルは、マシンセットに関連付けられた既存マシンおよびノードには伝播しません。

- 以下のコマンドは、ノードのラベルを追加または更新します。

```
$ oc label node <node> <key_1>=<value_1> ... <key_n>=<value_n>
```

以下に例を示します。

```
$ oc label nodes webconsole-7f7f6 unhealthy=true
```

- 以下のコマンドは、namespace 内のすべての Pod を更新します。

```
$ oc label pods --all <key_1>=<value_1>
```

以下に例を示します。

```
$ oc label pods --all status=unhealthy
```

5.2.3. ノードをスケジュール対象外 (Unschedulable) またはスケジュール対象 (Schedulable) としてマークする方法

デフォルトで、**Ready** ステータスの正常なノードはスケジュール対象としてマークされます。つまり、新規 Pod をこのノードに配置することができます。手動でノードをスケジュール対象外としてマークすると、新規 Pod のノードでのスケジュールがブロックされます。ノード上の既存 Pod には影響がありません。

- 以下のコマンドは、ノードをスケジュール対象外としてマークします。

出力例

```
$ oc adm cordon <node>
```

以下に例を示します。

```
$ oc adm cordon node1.example.com
```

出力例

```
node/node1.example.com cordoned

NAME                LABELS                                        STATUS
node1.example.com   kubernetes.io/hostname=node1.example.com
Ready,SchedulingDisabled
```

- 以下のコマンドは、現時点でスケジュール対象外のノードをスケジュール対象としてマークします。

```
$ oc adm uncordon <node1>
```

または、特定のノード名 (たとえば **<node>**) を指定する代わりに、**--selector=<node_selector>** オプションを使用して選択したノードをスケジュール対象またはスケジュール対象外としてマークすることができます。

5.2.4. スケジュール対象としてのコントロールプレーンノードの設定

コントロールプレーンノード (別名マスターノード) をスケジュール対象 (Schedulable) に設定できます。つまり、新規 Pod はコントロールプレーンノードに配置できるようになりました。デフォルトでは、コントロールプレーンノードはスケジュール対象ではありません。

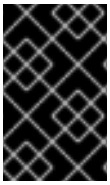
マスターをスケジュール対象 (Schedulable) に設定できますが、ワーカーノードを保持する必要があります。

注記

ワーカーノードのない OpenShift Container Platform をベアメタルクラスターにデプロイできます。この場合、コントロールプレーンノードはデフォルトでスケジュール対象としてマークされます。

mastersSchedulable フィールドを設定することで、コントロールプレーンノードをスケジュール対象として許可または禁止できます。

+



重要

コントロールプレーンノードをデフォルトのスケジュール不可からスケジュール可に設定するには、追加のサブスクリプションが必要です。これは、コントロールプレーンノードがワーカーノードになるためです。

手順

1. **schedulers.config.openshift.io** リソースを編集します。

```
$ oc edit schedulers.config.openshift.io cluster
```

2. **mastersSchedulable** フィールドを設定します。

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  creationTimestamp: "2019-09-10T03:04:05Z"
  generation: 1
  name: cluster
  resourceVersion: "433"
  selfLink: /apis/config.openshift.io/v1/schedulers/cluster
  uid: a636d30a-d377-11e9-88d4-0a60097bee62
spec:
  mastersSchedulable: false (1)
  policy:
    name: ""
status: {}
```

(1) コントロールプレーンノードがスケジュール対象 (Schedulable) になることを許可する場合は **true** に設定し、コントロールプレーンノードがスケジュール対象になることを拒否する場合は、**false** に設定します。

3. 変更を適用するためにファイルを保存します。

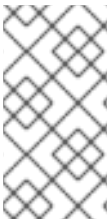
5.2.5. ノードの削除

5.2.5.1. クラスターからのノードの削除

CLI を使用してノードを削除する場合、ノードオブジェクトは Kubernetes で削除されますが、ノード自体にある Pod は削除されません。レプリケーションコントローラーで管理されないベア Pod は、OpenShift Container Platform からはアクセスできなくなります。レプリケーションコントローラーで管理されるベア Pod は、他の利用可能なノードに再スケジュールされます。ローカルのマニフェスト Pod は削除する必要があります。

手順

OpenShift Container Platform クラスターからノードを削除するには、適切な **MachineSet** オブジェクトを編集します。



注記

ベアメタルでクラスターを実行している場合、**MachineSet** オブジェクトを編集してノードを削除することはできません。マシンセットは、クラスターがクラウドプロバイダーに統合されている場合にのみ利用できます。代わりに、ノードを手作業で削除する前に、ノードをスケジュール解除し、ドレイン (解放) する必要があります。

1. クラスターにあるマシンセットを表示します。

```
$ oc get machinesets -n openshift-machine-api
```

   マシンセットは <clusterid>-worker-<aws-region-az> の形式で一覧表示されます。

2. マシンセットをスケーリングします。

```
$ oc scale --replicas=2 machineset <machineset> -n openshift-machine-api
```

マシンセットを使用してクラスターをスケーリングする方法の詳細は、**マシンセットの手動によるスケーリング** を参照してください。

5.2.5.2. ベアメタルクラスターからのノードの削除

CLI を使用してノードを削除する場合、ノードオブジェクトは Kubernetes で削除されますが、ノード自体にある Pod は削除されません。レプリケーションコントローラーで管理されないベア Pod は、OpenShift Container Platform からはアクセスできなくなります。レプリケーションコントローラーで管理されるベア Pod は、他の利用可能なノードに再スケジュールされます。ローカルのマニフェスト Pod は削除する必要があります。

手順

以下の手順を実行して、ベアメタルで実行されている OpenShift Container Platform クラスターからノードを削除します。

1. ノードにスケジュール対象外 (unschedulable) のマークを付けます。

```
$ oc adm cordon <node_name>
```

2. ノード上のすべての Pod をドレイン (解放) します。

```
$ oc adm drain <node_name> --force=true
```

このステップは、ノードがオフラインまたは応答しない場合に失敗する可能性があります。ノードが応答しない場合でも、共有ストレージに書き込むワークロードを実行している可能性があります。データの破損を防ぐには、続行する前に物理ハードウェアの電源を切ります。

3. クラスターからノードを削除します。

```
$ oc delete node <node_name>
```

ノードオブジェクトはクラスターから削除されていますが、これは再起動後や kubelet サービスが再起動される場合にクラスターに再び参加することができます。ノードとそのすべてのデータを永続的に削除するには、ノードの使用を停止 する必要があります。

4. 物理ハードウェアを電源を切っている場合は、ノードがクラスターに再度加わるように、そのハードウェアを再びオンに切り替えます。

5.2.6. SELinux ブール値の設定

OpenShift Container Platform を使用すると、Red Hat Enterprise Linux CoreOS(RHCOS) ノードで SELinux ブール値を有効または無効にできます。次の手順では、Machine Config Operator(MCO) を使用してノード上の SELinux ブール値を変更する方法について説明します。この手順では、ブール値の例として **container_manage_cgroup** を使用します。この値は、必要なブール値に変更できます。

前提条件

- OpenShift CLI (oc) がインストールされている。

手順

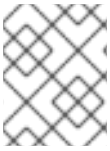
1. 次の例に示すように、**MachineConfig** オブジェクトを使用して新しい YAML ファイルを作成します。

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker
  name: 99-worker-setsebool
spec:
  config:
    ignition:
      version: 2.2.0
    systemd:
      units:
      - contents: |
          [Unit]
          Description=Set SELinux booleans
          Before=kubelet.service

          [Service]
          Type=oneshot
          ExecStart=/sbin/setsebool container_manage_cgroup=on
```

```
    RemainAfterExit=true

    [Install]
    WantedBy=multi-user.target graphical.target
enabled: true
name: setsebool.service
```

2. 次のコマンドを実行して、新しい **MachineConfig** オブジェクトを作成します。

```
$ oc create -f 99-worker-setsebool.yaml
```



注記

MachineConfig オブジェクトに変更を適用すると、変更が適用された後、影響を受けるすべてのノードが正常に再起動します。

5.2.7. カーネル引数のノードへの追加

特殊なケースとして、クラスターのノードセットにカーネル引数を追加する必要がある場合があります。これは十分に注意して実行する必要があり、設定する引数による影響を十分に理解している必要があります。

警告

カーネル引数を正しく使用しないと、システムが起動不可能になる可能性があります。

設定可能なカーネル引数の例には、以下が含まれます。

- **enforcing=0**: SELinux (Security Enhanced Linux) を Permissive モードで実行するように設定します。Permissive モードでは、システムは、SELinux が読み込んだセキュリティーポリシーを実行しているかのように動作します。これには、オブジェクトのラベル付けや、アクセスを拒否したエントリーをログに出力するなどの動作が含まれますが、いずれの操作も拒否される訳ではありません。Permissive モードは、実稼働システムでの使用はサポートされませんが、デバッグには役に立ちます。

- **nosmt**: カーネルの対称マルチスレッド (SMT) を無効にします。マルチスレッドは、各 CPU の複数の論理スレッドを許可します。潜在的なクロススレッド攻撃に関連するリスクを減らすために、マルチテナント環境での **nosmt** の使用を検討できます。SMT を無効にすることは、基本的にパフォーマンスよりもセキュリティーを重視する選択をしていることになります。

カーネル引数の一覧と説明については、Kernel.org カーネルパラメーター を参照してください。

次の手順では、以下を特定する **MachineConfig** オブジェクトを作成します。

- カーネル引数を追加する一連のマシン。この場合、ワーカーロールを持つマシン。

- 既存のカーネル引数の最後に追加されるカーネル引数。

- マシン設定の一覧で変更が適用される場所を示すラベル。

前提条件

- 作業用の OpenShift Container Platform クラスターに対する管理者権限が必要です。

手順

1. OpenShift Container Platform クラスターの既存の **MachineConfig** を一覧表示し、マシン設定にラベルを付ける方法を判別します。

```
$ oc get MachineConfig
```

出力例

```
NAME                                               GENERATEDBYCONTROLLER
IGNITIONVERSION   AGE
00-master                                          52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0
33m
00-worker                                          52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0
33m
01-master-container-runtime                              52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
01-master-kubelet                                  52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
01-worker-container-runtime                              52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
01-worker-kubelet                                  52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
99-master-generated-registries                           52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
99-master-ssh                                                                 3.2.0             40m
99-worker-generated-registries                           52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
99-worker-ssh                                                                 3.2.0             40m
rendered-master-23e785de7587df95a4b517e0647e5ab7
52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0             33m
rendered-worker-5d596d9293ca3ea80c896a1191735bb1
52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0             33m
```

2. カーネル引数を識別する **MachineConfig** オブジェクトファイルを作成します (例: **05-worker-kernelarg-selinuxpermissive.yaml**)。

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker (1)
  name: 05-worker-kernelarg-selinuxpermissive (2)
spec:
  config:
    ignition:
      version: 3.2.0
  kernelArguments:
    - enforcing=0 (3)
```

1. 新しいカーネル引数をワーカーノードのみに適用します。
2. マシン設定 (05) 内の適切な場所を特定するための名前が指定されます (SELinux permissive モードを設定するためにカーネル引数を追加します)。
3. 正確なカーネル引数を **enforcing=0** として特定します。

3. 新規のマシン設定を作成します。

```
$ oc create -f 05-worker-kernelarg-selinuxpermissive.yaml
```

4. マシン設定で新規の追加内容を確認します。

```
$ oc get MachineConfig
```

出力例

```
NAME                                               GENERATEDBYCONTROLLER
IGNITIONVERSION   AGE
00-master                                          52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0
33m
00-worker                                          52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0
33m
01-master-container-runtime                              52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
01-master-kubelet                                        52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
01-worker-container-runtime                              52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
01-worker-kubelet                                        52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
05-worker-kernelarg-selinuxpermissive                                              3.2.0             105s
99-master-generated-registries                           52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
99-master-ssh                                                                      3.2.0             40m
99-worker-generated-registries                           52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
99-worker-ssh                                                                      3.2.0             40m
rendered-master-23e785de7587df95a4b517e0647e5ab7
52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0             33m
rendered-worker-5d596d9293ca3ea80c896a1191735bb1
52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0             33m
```

5. ノードを確認します。

```
$ oc get nodes
```

出力例

```
NAME                         STATUS                     ROLES    AGE   VERSION
ip-10-0-136-161.ec2.internal   Ready                      worker   28m   v1.20.0
ip-10-0-136-243.ec2.internal   Ready                      master   34m   v1.20.0
ip-10-0-141-105.ec2.internal   Ready,SchedulingDisabled   worker   28m   v1.20.0
```

```
ip-10-0-142-249.ec2.internal   Ready                 master   34m   v1.20.0
ip-10-0-153-11.ec2.internal    Ready                 worker   28m   v1.20.0
ip-10-0-153-150.ec2.internal   Ready                 master   34m   v1.20.0
```

変更が適用されているため、各ワーカーノードのスケジューリングが無効にされていることを確認できます。

6. ワーカーノードのいずれかに移動し、カーネルコマンドライン引数 (ホストの **/proc/cmdline** 内) を一覧表示して、カーネル引数が機能することを確認します。

```
$ oc debug node/ip-10-0-141-105.ec2.internal
```

出力例

```
Starting pod/ip-10-0-141-105ec2internal-debug ...
To use host binaries, run `chroot /host`

sh-4.2# cat /host/proc/cmdline
BOOT_IMAGE=/ostree/rhcos-... console=tty0 console=ttyS0,115200n8
rootflags=defaults,prjquota rw root=UUID=fd0... ostree=/ostree/boot.0/rhcos/16...
coreos.oem.id=qemu coreos.oem.id=ec2 ignition.platform.id=ec2 enforcing=0

sh-4.2# exit
```

enforcing=0 引数が他のカーネル引数に追加されていることを確認できるはずです。

5.2.8. 関連情報

- MachineSet を使用してクラスターをスケーリングする方法の詳細は、MachineSet の手動によるスケーリング を参照してください。

5.3. ノードの管理

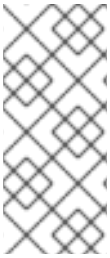
OpenShift Container Platform は、KubeletConfig カスタムリソース (CR) を使ってノードの設定を管理します。**KubeletConfig** オブジェクトのインスタンスを作成すると、管理対象のマシン設定がノードの設定を上書きするために作成されます。

注記

リモートマシンにログインして設定を変更する方法はサポートされていません。

5.3.1. ノードの変更

クラスターまたはマシンプールの設定を変更するには、カスタムリソース定義 (CRD) または **kubeletConfig** オブジェクトを作成する必要があります。OpenShift Container Platform は、Machine Config Controller を使って、変更をクラスターに適用するために CRD を使用して導入された変更を監視します。



注記

kubeletConfig オブジェクトのフィールドは、アップストリームの Kubernetes から kubelet に直接渡されるため、これらのフィールドの検証は kubelet 自体によって直接処理されます。これらのフィールドの有効な値については、関連する Kubernetes のドキュメントを参照してください。**kubeletConfig** オブジェクトの値が無効な場合、クラスターノードが使用できなくなる可能性があります。

手順

1. 設定する必要のあるノードタイプの静的な CRD、Machine Config Pool に関連付けられたラベルを取得します。以下のいずれかの手順を実行します。

   a. 必要なマシン設定プールの現在のラベルをチェックします。
   以下に例を示します。

   ```
   $  oc get machineconfigpool  --show-labels
   ```

   出力例

   ```
   NAME      CONFIG                                         UPDATED   UPDATING   DEGRADED
   LABELS
   master    rendered-master-e05b81f5ca4db1d249a1bf32f9ec24fd   True      False
   False     operator.machineconfiguration.openshift.io/required-for-upgrade=
   worker    rendered-worker-f50e78e1bc06d8e82327763145bfcf62   True      False
   False
   ```

   b. 必要なマシン設定プールにカスタムラベルを追加します。
   以下に例を示します。

   ```
   $ oc label machineconfigpool worker custom-kubelet=enabled
   ```

2. 設定の変更用に **kubeletconfig** カスタムリソース (CR) を作成します。
   以下に例を示します。

   custom-config CR の設定例

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: KubeletConfig
   metadata:
     name: custom-config (1)
   spec:
     machineConfigPoolSelector:
       matchLabels:
         custom-kubelet: enabled (2)
     kubeletConfig: (3)
       podsPerCore: 10
       maxPods: 250
       systemReserved:
         cpu: 2000m
         memory: 1Gi
   ```

   (1) CR に名前を割り当てます。

2 設定変更を適用するラベルを指定します。これは、マシン設定プールに追加するラベルになります。

3 変更する必要のある新しい値を指定します。

3. CR オブジェクトを作成します。

```
$ oc create -f <file-name>
```

以下に例を示します。

```
$ oc create -f master-kube-config.yaml
```

ほとんどの Kubelet 設定オプション はユーザーが設定できます。以下のオプションは上書きが許可されていません。

- CgroupDriver
- ClusterDNS
- ClusterDomain
- RuntimeRequestTimeout
- StaticPodPath

5.4. ノードあたりの POD の最大数の管理

OpenShift Container Platform では、ノードのプロセッサーコアの数に基づいて、ノードで実行可能な Pod の数、ハード制限、またはその両方を設定できます。両方のオプションを使用した場合、より低い値の方がノード上の Pod の数を制限します。

これらの値を超えると、以下の状態が生じる可能性があります。

- OpenShift Container Platform の CPU 使用率が増加
- Pod のスケジューリングの速度が遅くなる。
- (ノードのメモリー量によって) メモリー不足のシナリオが生じる可能性。
- IP アドレスプールが使い切られる。
- リソースのオーバーコミット、およびこれによるアプリケーションのパフォーマンスの低下。

注記

単一コンテナーを保持する Pod は実際には 2 つのコンテナーを使用します。2 つ目のコンテナーは実際のコンテナーの起動前にネットワークを設定します。その結果、10 の Pod を実行しているノードでは、実際には 20 のコンテナーが実行されていることになります。

podsPerCore パラメーターは、ノードのプロセッサーコア数に基づいてノードが実行できる Pod 数を制限します。たとえば、4 プロセッサーコアを搭載したノードで **podsPerCore** が **10** に設定されている場合、このノードで許可される Pod の最大数は 40 になります。

maxPods パラメーターは、ノードのプロパティーにかかわらず、ノードが実行できる Pod 数を固定値に制限します。

5.4.1. ノードあたりの Pod の最大数の設定

podsPerCore および **maxPods** の 2 つのパラメーターはノードに対してスケジュールできる Pod の最大数を制御します。両方のオプションを使用した場合、より低い値の方がノード上の Pod の数を制限します。

たとえば、**podsPerCore** が 4 つのプロセッサーコアを持つノード上で、**10** に設定されていると、ノード上で許容される Pod の最大数は 40 になります。

前提条件

1. 設定するノードタイプの静的な **MachineConfigPool** CRD に関連付けられたラベルを取得します。以下のいずれかの手順を実行します。

   a. マシン設定プールを表示します。

   ```
   $ oc describe machineconfigpool <name>
   ```

   以下に例を示します。

   ```
   $ oc describe machineconfigpool worker
   ```

   出力例

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: MachineConfigPool
   metadata:
     creationTimestamp: 2019-02-08T14:52:39Z
     generation: 1
     labels:
       custom-kubelet: small-pods 1
   ```

   1 ラベルが追加されると、**labels** の下に表示されます。

   b. ラベルが存在しない場合は、キー/値のペアを追加します。

   ```
   $ oc label machineconfigpool worker custom-kubelet=small-pods
   ```

手順

1. 設定変更のためのカスタムリソース (CR) を作成します。

   max-pods CR の設定例

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: KubeletConfig
   metadata:
     name: set-max-pods 1
   spec:
   ```

```
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: small-pods 2
  kubeletConfig:
    podsPerCore: 10 3
    maxPods: 250 4
```

1 CR に名前を割り当てます。

2 設定の変更を適用するラベルを指定します。

3 ノードがプロセッサーコアの数に基づいて実行できる Pod の数を指定します。

4 ノードのプロパティーにかかわらず、ノードが実行できる Pod 数を固定値に指定します。

注記

podsPerCore を **0** に設定すると、この制限が無効になります。

上記の例では、**podsPerCore** のデフォルト値は **10** であり、**maxPods** のデフォルト値は **250** です。つまり、ノードのコア数が 25 以上でない限り、デフォルトにより **podsPerCore** が制限要素になります。

2. 変更が適用されるかどうかを確認するために、**MachineConfigPool** CRD を一覧表示します。変更が Machine Config Controller によって取得されると、**UPDATING** 列で **True** と報告されます。

```
$ oc get machineconfigpools
```

出力例

```
NAME     CONFIG                    UPDATED   UPDATING   DEGRADED
master   master-9cc2c72f205e103bb534   False     False      False
worker   worker-8cecd1236b33ee3f8a5e   False     True       False
```

変更が完了すると、**UPDATED** 列で **True** と報告されます。

```
$ oc get machineconfigpools
```

出力例

```
NAME     CONFIG                    UPDATED   UPDATING   DEGRADED
master   master-9cc2c72f205e103bb534   False     True       False
worker   worker-8cecd1236b33ee3f8a5e   True      False      False
```

5.5. NODE TUNING OPERATOR の使用

Node Tuning Operator に関する説明のほか、この Operator を使用して Tuned デーモンのオーケストレーションを実行し、ノードレベルのチューニングを管理する方法についても説明します。

Node Tuning Operator は、Tuned デーモンのオーケストレーションによるノードレベルのチューニン

グの管理に役立ちます。ほとんどの高パフォーマンスアプリケーションでは、一定レベルのカーネルのチューニングが必要です。Node Tuning Operator は、ノードレベルの sysctl の統一された管理インターフェイスをユーザーに提供し、ユーザーが指定するカスタムチューニングを追加できるよう柔軟性を提供します。

Operator は、コンテナー化された OpenShift Container Platform の Tuned デーモンを Kubernetes デーモンセットとして管理します。これにより、カスタムチューニング仕様が、デーモンが認識する形式でクラスターで実行されるすべてのコンテナー化された Tuned デーモンに渡されます。デーモンは、ノードごとに 1 つずつ、クラスターのすべてのノードで実行されます。

コンテナー化された Tuned デーモンによって適用されるノードレベルの設定は、プロファイルの変更をトリガーするイベントで、または終了シグナルの受信および処理によってコンテナー化された Tuned デーモンが正常に終了する際にロールバックされます。

Node Tuning Operator は、バージョン 4.1 以降における標準的な OpenShift Container Platform インストールの一部となっています。

5.5.1. Node Tuning Operator 仕様サンプルへのアクセス

このプロセスを使用して Node Tuning Operator 仕様サンプルにアクセスします。

手順

1. 以下を実行します。

```
$ oc get Tuned/default -o yaml -n openshift-cluster-node-tuning-operator
```

デフォルトの CR は、OpenShift Container Platform プラットフォームの標準的なノードレベルのチューニングを提供することを目的としており、Operator 管理の状態を設定するためにのみ変更できます。デフォルト CR へのその他のカスタム変更は、Operator によって上書きされます。カスタムチューニングの場合は、独自のチューニングされた CR を作成します。新規に作成された CR は、ノード/Pod ラベルおよびプロファイルの優先順位に基づいて OpenShift Container Platform ノードに適用されるデフォルトの CR およびカスタムチューニングと組み合わされます。

警告

特定の状況で Pod ラベルのサポートは必要なチューニングを自動的に配信する便利な方法ですが、この方法は推奨されず、とくに大規模なクラスターにおいて注意が必要です。デフォルトの調整された CR は Pod ラベル一致のない状態で提供されます。カスタムプロファイルが Pod ラベル一致のある状態で作成される場合、この機能はその時点で有効になります。Pod ラベル機能は、Node Tuning Operator の今後のバージョンで非推奨になる場合があります。

5.5.2. カスタムチューニング仕様

Operator のカスタムリソース (CR) には 2 つの重要なセクションがあります。1 つ目のセクションの **profile:** は Tuned プロファイルおよびそれらの名前の一覧です。2 つ目の **recommend:** は、プロファイル選択ロジックを定義します。

複数のカスタムチューニング仕様は、Operator の namespace に複数の CR として共存できます。新規 CR の存在または古い CR の削除は Operator によって検出されます。既存のカスタムチューニング仕様はすべてマージされ、コンテナー化された Tuned デーモンの適切なオブジェクトは更新されます。

管理状態

Operator 管理の状態は、デフォルトの Tuned CR を調整して設定されます。デフォルトで、Operator は Managed 状態であり、**spec.managementState** フィールドはデフォルトの Tuned CR に表示されません。Operator Management 状態の有効な値は以下のとおりです。

- Managed: Operator は設定リソースが更新されるとそのオペランドを更新します。
- Unmanaged: Operator は設定リソースへの変更を無視します。
- Removed: Operator は Operator がプロビジョニングしたオペランドおよびリソースを削除します。

プロファイルデータ

profile: セクションは、Tuned プロファイルおよびそれらの名前を一覧表示します。

```
profile:
- name: tuned_profile_1
  data: |
    # Tuned profile specification
    [main]
    summary=Description of tuned_profile_1 profile

    [sysctl]
    net.ipv4.ip_forward=1
    # ... other sysctl's or other Tuned daemon plugins supported by the containerized Tuned

# ...

- name: tuned_profile_n
  data: |
    # Tuned profile specification
    [main]
    summary=Description of tuned_profile_n profile

    # tuned_profile_n profile settings
```

推奨プロファイル

profile: 選択ロジックは、CR の **recommend:** セクションによって定義されます。**recommend:** セクションは、選択基準に基づくプロファイルの推奨項目の一覧です。

```
recommend:
<recommend-item-1>
# ...
<recommend-item-n>
```

一覧の個別項目:

```
- machineConfigLabels: 1
```

```
  <mcLabels> 2
match: 3
  <match> 4
priority: <priority> 5
profile: <tuned_profile_name> 6
operand: 7
  debug: <bool> 8
```

1 オプション:

2 キー/値の **MachineConfig** ラベルのディクショナリー。キーは一意である必要があります。

3 省略する場合は、優先度の高いプロファイルが最初に一致するか、または **machineConfigLabels** が設定されていない限り、プロファイルの一致が想定されます。

4 オプションの一覧。

5 プロファイルの順序付けの優先度。数値が小さいほど優先度が高くなります (**0** が最も高い優先度になります)。

6 一致に適用する TuneD プロファイル。例: **tuned_profile_1**

7 オプションのオペランド設定。

8 TuneD デーモンのデバッグオンまたはオフを有効にします。オプションは、オンの場合は **true**、オフの場合は **false** です。デフォルトは **false** です。

<match> は、以下のように再帰的に定義されるオプションの一覧です。

```
- label: <label_name> 1
  value: <label_value> 2
  type: <label_type> 3
    <match> 4
```

1 ノードまたは Pod のラベル名。

2 オプションのノードまたは Pod のラベルの値。省略されている場合も、**<label_name>** があるだけで一致条件を満たします。

3 オプションのオブジェクトタイプ (**node** または **pod**)。省略されている場合は、**node** が想定されます。

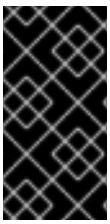
4 オプションの **<match>** 一覧。

<match> が省略されない場合、ネストされたすべての **<match>** セクションが **true** に評価される必要もあります。そうでない場合には **false** が想定され、それぞれの **<match>** セクションのあるプロファイルは適用されず、推奨されません。そのため、ネスト化 (子の **<match>** セクション) は論理 AND 演算子として機能します。これとは逆に、**<match>** 一覧のいずれかの項目が一致する場合、**<match>** の一覧全体が **true** に評価されます。そのため、一覧は論理 OR 演算子として機能します。

machineConfigLabels が定義されている場合、マシン設定プールベースのマッチングが指定の **recommend:** 一覧の項目に対してオンになります。**<mcLabels>** はマシン設定のラベルを指定します。マシン設定は、プロファイル **<tuned_profile_name>** についてカーネル起動パラメーターなどのホ

スト設定を適用するために自動的に作成されます。この場合、マシン設定セレクターが **<mcLabels>** に一致するすべてのマシン設定プールを検索し、プロファイル **<tuned_profile_name>** を確認されるマシン設定プールが割り当てられるすべてのノードに設定する必要があります。マスターロールとワーカーのロールの両方を持つノードをターゲットにするには、マスターロールを使用する必要があります。

一覧項目の **match** および **machineConfigLabels** は論理 OR 演算子によって接続されます。 **match** 項目は、最初にショートサーキット方式で評価されます。そのため、**true** と評価される場合、**machineConfigLabels** 項目は考慮されません。



重要

マシン設定プールベースのマッチングを使用する場合、同じハードウェア設定を持つノードを同じマシン設定プールにグループ化することが推奨されます。この方法に従わない場合は、チューニングされたオペランドが同じマシン設定プールを共有する 2 つ以上のノードの競合するカーネルパラメーターを計算する可能性があります。

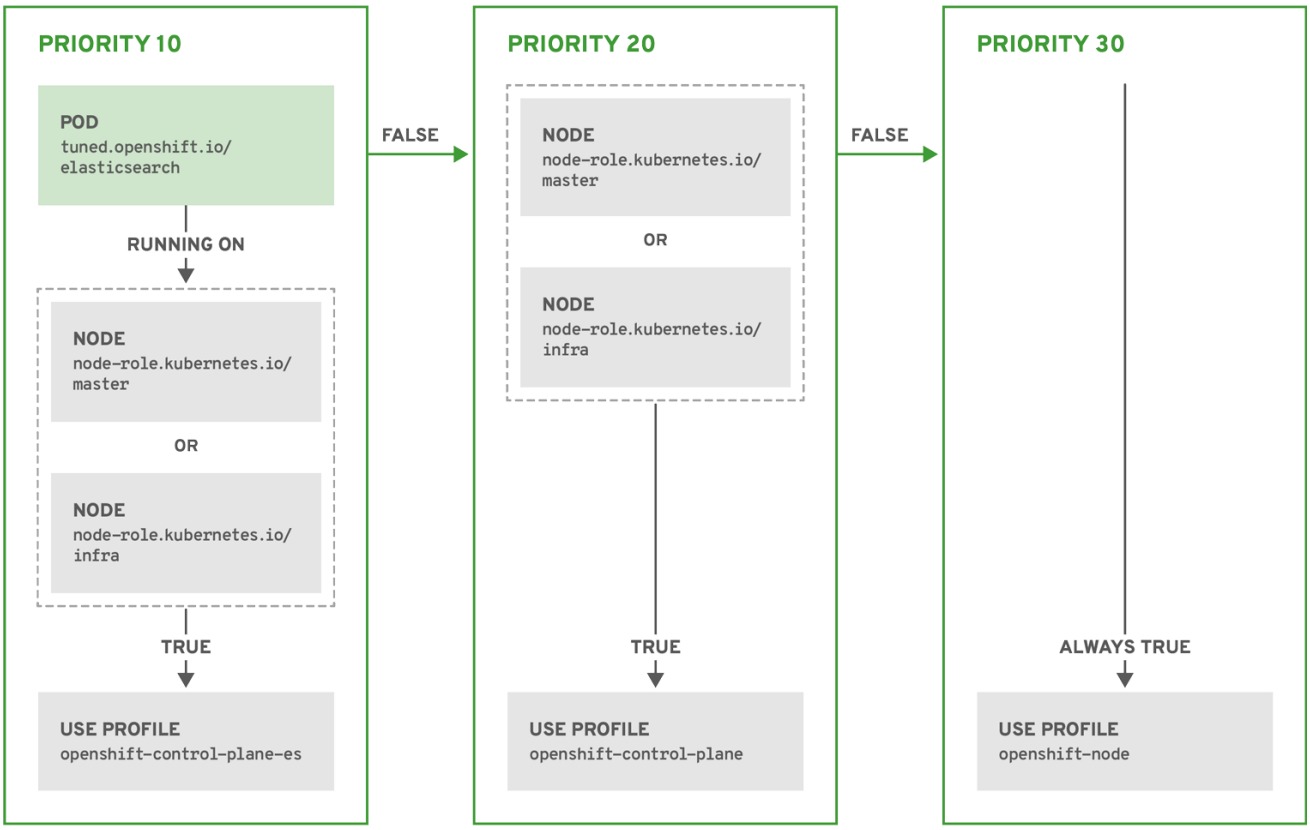
例: ノード/Pod ラベルベースのマッチング

```
- match:
  - label: tuned.openshift.io/elasticsearch
    match:
    - label: node-role.kubernetes.io/master
    - label: node-role.kubernetes.io/infra
    type: pod
  priority: 10
  profile: openshift-control-plane-es
- match:
  - label: node-role.kubernetes.io/master
  - label: node-role.kubernetes.io/infra
  priority: 20
  profile: openshift-control-plane
- priority: 30
  profile: openshift-node
```

上記のコンテナー化された Tuned デーモンの CR は、プロファイルの優先順位に基づいてその **recommend.conf** ファイルに変換されます。最も高い優先順位 (**10**) を持つプロファイルは **openshift-control-plane-es** であるため、これが最初に考慮されます。指定されたノードで実行されるコンテナー化された Tuned デーモンは、同じノードに **tuned.openshift.io/elasticsearch** ラベルが設定された Pod が実行されているかどうかを確認します。これがない場合、 **<match>** セクション全体が **false** として評価されます。このラベルを持つこのような Pod がある場合、 **<match>** セクションが **true** に評価されるようにするには、ノードラベルは **node-role.kubernetes.io/master** または **node-role.kubernetes.io/infra** である必要もあります。

優先順位が **10** のプロファイルのラベルが一致した場合、**openshift-control-plane-es** プロファイルが適用され、その他のプロファイルは考慮されません。ノード/Pod ラベルの組み合わせが一致しない場合、2 番目に高い優先順位プロファイル (**openshift-control-plane**) が考慮されます。このプロファイルは、コンテナー化されたチューニング済み Pod が **node-role.kubernetes.io/master** または **node-role.kubernetes.io/infra** ラベルを持つノードで実行される場合に適用されます。

最後に、プロファイル **openshift-node** には最低の優先順位である **30** が設定されます。これには **<match>** セクションがないため、常に一致します。これは、より高い優先順位の他のプロファイルが指定されたノードで一致しない場合に **openshift-node** プロファイルを設定するために、最低の優先順位のノードが適用される汎用的な (catch-all) プロファイルとして機能します。



例: マシン設定プールベースのマッチング

```
apiVersion: tuned.openshift.io/v1
kind: Tuned
metadata:
  name: openshift-node-custom
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile:
  - data: |
      [main]
      summary=Custom OpenShift node profile with an additional kernel parameter
      include=openshift-node
      [bootloader]
      cmdline_openshift_node_custom=+skew_tick=1
    name: openshift-node-custom

  recommend:
  - machineConfigLabels:
      machineconfiguration.openshift.io/role: "worker-custom"
    priority: 20
    profile: openshift-node-custom
```

ノードの再起動を最小限にするには、ターゲットノードにマシン設定プールのノードセレクターが一致するラベルを使用してラベルを付け、上記の Tuned CR を作成してから、最後にカスタムのマシン設定プール自体を作成します。

5.5.3. クラスターに設定されるデフォルトのプロファイル

以下は、クラスターに設定されるデフォルトのプロファイルです。

```
apiVersion: tuned.openshift.io/v1
kind: Tuned
metadata:
  name: default
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile:
  - name: "openshift"
    data: |
      [main]
      summary=Optimize systems running OpenShift (parent profile)
      include=${f:virt_check:virtual-guest:throughput-performance}

      [selinux]
      avc_cache_threshold=8192

      [net]
      nf_conntrack_hashsize=131072

      [sysctl]
      net.ipv4.ip_forward=1
      kernel.pid_max=>4194304
      net.netfilter.nf_conntrack_max=1048576
      net.ipv4.conf.all.arp_announce=2
      net.ipv4.neigh.default.gc_thresh1=8192
      net.ipv4.neigh.default.gc_thresh2=32768
      net.ipv4.neigh.default.gc_thresh3=65536
      net.ipv6.neigh.default.gc_thresh1=8192
      net.ipv6.neigh.default.gc_thresh2=32768
      net.ipv6.neigh.default.gc_thresh3=65536
      vm.max_map_count=262144

      [sysfs]
      /sys/module/nvme_core/parameters/io_timeout=4294967295
      /sys/module/nvme_core/parameters/max_retries=10

  - name: "openshift-control-plane"
    data: |
      [main]
      summary=Optimize systems running OpenShift control plane
      include=openshift

      [sysctl]
      # ktune sysctl settings, maximizing i/o throughput
      #
      # Minimal preemption granularity for CPU-bound tasks:
      # (default: 1 msec#  (1 + ilog(ncpus)), units: nanoseconds)
      kernel.sched_min_granularity_ns=10000000
      # The total time the scheduler will consider a migrated process
      # "cache hot" and thus less likely to be re-migrated
      # (system default is 500000, i.e. 0.5 ms)
      kernel.sched_migration_cost_ns=5000000
      # SCHED_OTHER wake-up granularity.
      #
```

```
    # Preemption granularity when tasks wake up.  Lower the value to
    # improve wake-up latency and throughput for latency critical tasks.
    kernel.sched_wakeup_granularity_ns=4000000

- name: "openshift-node"
  data: |
    [main]
    summary=Optimize systems running OpenShift nodes
    include=openshift

    [sysctl]
    net.ipv4.tcp_fastopen=3
    fs.inotify.max_user_watches=65536
    fs.inotify.max_user_instances=8192

recommend:
- profile: "openshift-control-plane"
  priority: 30
  match:
  - label: "node-role.kubernetes.io/master"
  - label: "node-role.kubernetes.io/infra"

- profile: "openshift-node"
  priority: 40
```

5.5.4. サポートされている Tuned デーモンプラグイン

[main] セクションを除き、以下の Tuned プラグインは、Tuned CR の **profile:** セクションで定義されたカスタムプロファイルを使用する場合にサポートされます。

- audio
- cpu
- disk
- eeepc_she
- modules
- mounts
- net
- scheduler
- scsi_host
- selinux
- sysctl
- sysfs
- usb

- video
- vm

これらのプラグインの一部によって提供される動的チューニング機能の中に、サポートされていない機能があります。以下の Tuned プラグインは現時点でサポートされていません。

- bootloader
- script
- systemd

詳細は、利用可能な Tuned プラグイン および Tuned の使用 を参照してください。

5.6. ノードの再起動について

プラットフォームで実行されているアプリケーションを停止せずにノードを再起動するには、まず Pod の退避を実行することが重要です。ルーティング階層によって可用性が高くなっている Pod については、何も実行する必要はありません。ストレージ (通常はデータベース) を必要とするその他の Pod については、1 つの Pod が一時的にオフラインになってもそれらの Pod が作動状態を維持できることを確認する必要があります。ステートフルな Pod の回復性はアプリケーションごとに異なりますが、いずれの場合でも、ノードの非アフィニティー (node anti-affinity) を使用して Pod が使用可能なノードにわたって適切に分散するようにスケジューラーを設定することが重要になります。

別の課題として、ルーターやレジストリーのような重要なインフラストラクチャーを実行しているノードを処理する方法を検討する必要があります。同じノードの退避プロセスが適用されますが、一部のエッジケースについて理解しておくことが重要です。

5.6.1. 重要なインフラストラクチャーを実行するノードの再起動について

ルーター Pod、レジストリー Pod、モニターリング Pod などの重要な OpenShift Container Platform インフラストラクチャーコンポーネントをホストするノードを再起動する場合、これらのコンポーネントを実行するために少なくとも 3 つのノードが利用可能であることを確認します。

以下のシナリオは、2 つのノードのみが利用可能な場合に、どのように OpenShift Container Platform で実行されているアプリケーションでサービスの中断が生じ得るかを示しています。

- ノード A がスケジュール対象外としてマークされており、すべての Pod の退避が行われている。
- このノードで実行されているレジストリー Pod がノード B に再デプロイされる。 ノード B が両方のレジストリー Pod を実行しています。
- ノード B はスケジュール対象外としてマークされ、退避が行われる。
- ノード B の 2 つの Pod エンドポイントを公開するサービスは、それらがノード A に再デプロイされるまでの短い期間にすべてのエンドポイントを失う。

インフラストラクチャーコンポーネントの 3 つのノードを使用する場合、このプロセスではサービスの中断が生じません。しかし、Pod のスケジューリングにより、退避してローテーションに戻される最後のノードにはレジストリー Pod がありません。他のノードのいずれかには 2 つのレジストリー Pod があります。3 番目のレジストリー Pod を最後のノードでスケジュールするには、Pod の非アフィニティーを使用してスケジューラーが同じノード上で 2 つのレジストリー Pod を見つけるのを防ぎます。

関連情報

- Pod の非アフィニティーについての詳細は、 アフィニティールールと非アフィニティールールの使用による他の Pod との相対での Pod の配置 について参照してください。

5.6.2. Pod の非アフィニティーを使用するノードの再起動

Pod の非アフィニティーは、ノードの非アフィニティーとは若干異なります。ノードの非アフィニティーの場合、Pod のデプロイ先となる適切な場所が他にない場合には違反が生じる可能性があります。Pod の非アフィニティーの場合は required (必須) または preferred (優先) のいずれかに設定できます。

これが有効になっていると、2 つのインフラストラクチャーノードのみが利用可能で、1 つのノードが再起動される場合に、コンテナーイメージレジストリー Pod は他のノードで実行できなくなります。**oc get pods** は、適切なノードが利用可能になるまで Pod を Unready (準備が未完了) として報告します。ノードが利用可能になり、すべての Pod が Ready (準備ができている) 状態に戻ると、次のノードを再起動することができます。

手順

Pod の非アフィニティーを使用してノードを再起動するには、以下の手順を実行します。

1. ノードの仕様を編集して Pod の非アフィニティーを設定します。

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-antiaffinity
spec:
  affinity:
    podAntiAffinity: 1
      preferredDuringSchedulingIgnoredDuringExecution: 2
      - weight: 100 3
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: registry 4
              operator: In 5
              values:
              - default
          topologyKey: kubernetes.io/hostname
```

1. Pod の非アフィニティーを設定するためのスタンザです。
2. preferred (優先) ルールを定義します。
3. preferred (優先) ルールの重みを指定します。最も高い重みを持つノードが優先されます。
4. 非アフィニティールールが適用される時を決定する Pod ラベルの説明です。ラベルのキーおよび値を指定します。
5. 演算子は、既存 Pod のラベルと新規 Pod の仕様の **matchExpression** パラメーターの値のセットの間の関係を表します。これには **In**、**NotIn**、**Exists**、または **DoesNotExist** のいずれかを使用できます。

この例では、コンテナーイメージレジストリー Pod に **registry=default** のラベルがあることを想定しています。Pod の非アフィニティーでは任意の Kubernetes の一致式を使用できます。

2. スケジューリングポリシーファイルで、**MatchInterPodAffinity** スケジューラー述語を有効にします。

3. ノードの正常な再起動を実行します。

5.6.3. ルーターを実行しているノードを再起動する方法について

ほとんどの場合、OpenShift Container Platform ルーターを実行している Pod はホストポートを公開します。

PodFitsPorts スケジューラー述語は、同じポートを使用するルーター Pod が同じノード上で実行できないようにし、Pod の非アフィニティーが確保されるようにします。ルーターが高可用性を確保するために IP フェイルオーバーに依存する場合は、他に必要な設定等はありません。

高可用性のための AWS Elastic Load Balancing のような外部サービスに依存するルーター Pod の場合は、ルーターの再起動に対応するサービスが必要になります。

ルーター Pod でホストのポートが設定されていないということも稀にあります。この場合は、インフラストラクチャーノードについての推奨される再起動プロセスに従う必要があります。

5.6.4. ノードを正常に再起動する

ノードを再起動する前に、ノードでのデータ損失を回避するために、etcd データをバックアップすることをお勧めします。

手順

ノードの正常な再起動を実行するには:

1. ノードにスケジュール対象外 (unschedulable) のマークを付けます。

```
$ oc adm cordon <node1>
```

2. ノードをドレインして、実行中のすべての Pod を削除します。

```
$ oc adm drain <node1> --ignore-daemonsets --delete-emptydir-data
```

カスタムの Pod の Disruption Budget (停止状態の予算、PDB) 関連付けられた Pod を退避できないというエラーが発生することがあります。

エラーの例

```
error when evicting pods/"rails-postgresql-example-1-72v2w" -n "rails" (will retry after 5s):
Cannot evict pod as it would violate the pod's disruption budget.
```

この場合、drain コマンドを再度実行し、**disable-eviction** フラグを追加し、PDB チェックを省略します。

```
$ oc adm drain <node1> --ignore-daemonsets --delete-emptydir-data --force --disable-
eviction
```

3. デバッグモードでノードにアクセスします。

```
$ oc debug node/<node1>
```

4. ルートディレクトリーをホストに切り替えます。

```
$ chroot /host
```

5. ノードを再起動します。

```
$ systemctl reboot
```

すぐに、ノードは **NotReady** 状態になります。

6. 再起動が完了したら、以下のコマンドを実行して、ノードをスケジューリング可能な状態にします。

```
$ oc adm uncordon <node1>
```

7. ノードの準備ができていることを確認します。

```
$ oc get node <node1>
```

出力例

```
NAME    STATUS  ROLES    AGE     VERSION
<node1> Ready   worker   6d22h   v1.18.3+b0068a8
```

関連情報

etcd データのバックアップの詳細については、Backing up etcd data を参照してください。

5.7. ガベージコレクションを使用しているノードリソースの解放

管理者は、OpenShift Container Platform を使用し、ガベージコレクションによってリソースを解放することにより、ノードを効率的に実行することができます。

OpenShift Container Platform ノードは、2 種類のガベージコレクションを実行します。

- コンテナーのガベージコレクション: 終了したコンテナーを削除します。
- イメージのガベージコレクション: 実行中のどの Pod からも参照されていないイメージを削除します。

5.7.1. 終了したコンテナーがガベージコレクションによって削除される仕組みについて

コンテナーのガベージコレクションは、エビクションしきい値を使用して実行することができます。

エビクションしきい値がガーベージコレクションに設定されていると、ノードは Pod のコンテナーが API から常にアクセス可能な状態になるよう試みます。Pod が削除された場合、コンテナーも削除されます。コンテナーは Pod が削除されず、エビクションしきい値に達していない限り保持されます。ノードがディスク不足 (disk pressure) の状態になっていると、コンテナーが削除され、それらのログは **oc logs** を使用してアクセスできなくなります。

- **eviction-soft** - ソフトエビクションのしきい値は、エビクションしきい値と要求される管理者指定の猶予期間を組み合わせます。
- **eviction-hard** - ハードエビクションのしきい値には猶予期間がなく、検知されると、OpenShift Container Platform はすぐにアクションを実行します。

以下の表は、エビクションしきい値の一覧です。

表5.2 コンテナーのガベージコレクションを設定するための変数

ノードの状態	エビクションシグナル	説明
MemoryPressure	**memory.available**	ノードで利用可能なメモリー。
DiskPressure	• **nodefs.available** • **nodefs.inodesFree** • **imagefs.available** • **imagefs.inodesFree**	ノードのルートファイルシステム (**nodefs**) またはイメージファイルシステム (**imagefs**) で利用可能なディスク領域または i ノード。

注記

evictionHard の場合、これらのパラメーターをすべて指定する必要があります。すべてのパラメーターを指定しないと、指定したパラメーターのみが適用され、ガベージコレクションが正しく機能しません。

ノードがソフトエビクションしきい値の上限と下限の間で変動し、その関連する猶予期間を超えていない場合、対応するノードは、**true** と **false** の間で常に変動します。したがって、スケジューラーは適切なスケジュールを決定できない可能性があります。

この変動から保護するには、**eviction-pressure-transition-period** フラグを使用して、OpenShift Container Platform が不足状態から移行するまでにかかる時間を制御します。OpenShift Container Platform は、false 状態に切り替わる前の指定された期間に、エビクションしきい値を指定された不足状態に一致するように設定しません。

5.7.2. イメージがガベージコレクションによって削除される仕組みについて

イメージのガベージコレクションでは、ノードの **cAdvisor** によって報告されるディスク使用量に基づいて、ノードから削除するイメージを決定します。

イメージのガベージコレクションのポリシーは、以下の 2 つの条件に基づいています。

- イメージのガベージコレクションをトリガーするディスク使用量のパーセント (整数で表される) です。デフォルトは 85 です。
- イメージのガベージコレクションが解放しようとするディスク使用量のパーセント (整数で表される) です。デフォルトは 80 です。

イメージのガベージコレクションのために、カスタムリソースを使用して、次の変数のいずれかを変更することができます。

表5.3 イメージのガベージコレクションを設定するための変数

設定	説明
imageMinimumGCAge	ガベージコレクションによって削除されるまでの未使用のイメージの有効期間。デフォルトは、2m です。
imageGCHighThresholdPercent	イメージのガベージコレクションをトリガーするディスク使用量のパーセント (整数で表される) です。デフォルトは 85 です。
imageGCLowThresholdPercent	イメージのガベージコレクションが解放しようとするディスク使用量のパーセント (整数で表される) です。デフォルトは 80 です。

以下の 2 つのイメージ一覧がそれぞれのガベージコレクターの実行で取得されます。

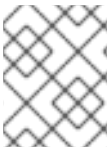
1. 1 つ以上の Pod で現在実行されているイメージの一覧
2. ホストで利用可能なイメージの一覧

新規コンテナーの実行時に新規のイメージが表示されます。すべてのイメージにはタイムスタンプのマークが付けられます。イメージが実行中 (上記の最初の一覧) か、または新規に検出されている (上記の 2 番目の一覧) 場合、これには現在の時間のマークが付けられます。残りのイメージには以前のタイムスタンプのマークがすでに付けられています。すべてのイメージはタイムスタンプで並び替えられます。

コレクションが開始されると、停止条件を満たすまでイメージが最も古いものから順番に削除されます。

5.7.3. コンテナーおよびイメージのガベージコレクションの設定

管理者は、**kubeletConfig** オブジェクトを各マシン設定プール用に作成し、OpenShift Container Platform によるガベージコレクションの実行方法を設定できます。



注記

OpenShift Container Platform は、各マシン設定プールの **kubeletConfig** オブジェクトを 1 つのみサポートします。

次のいずれかの組み合わせを設定できます。

- コンテナーのソフトエビクション
- コンテナーのハードエビクション
- イメージのエビクション

前提条件

1. 設定するノードタイプの静的な **MachineConfigPool** CRD に関連付けられたラベルを取得します。以下のいずれかの手順を実行します。
   a. マシン設定プールを表示します。

```
$ oc describe machineconfigpool <name>
```

以下に例を示します。

```
$ oc describe machineconfigpool worker
```

出力例

```
Name:         worker
Namespace:
Labels:       custom-kubelet=small-pods 1
```

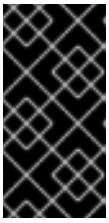
1 ラベルが追加されると、**Labels** の下に表示されます。

b. ラベルが存在しない場合は、キー/値のペアを追加します。

```
$ oc label machineconfigpool worker custom-kubelet=small-pods
```

手順

1. 設定変更のためのカスタムリソース (CR) を作成します。



重要

ファイルシステムが 1 つの場合、または **/var/lib/kubelet** と **/var/lib/containers/** が同じファイルシステムにある場合、最も大きな値の設定が満たされるとエビクションがトリガーされます。ファイルシステムはエビクションをトリガーします。

コンテナーのガベージコレクション CR のサンプル設定:

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: worker-kubeconfig 1
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: small-pods 2
  kubeletConfig:
    evictionSoft: 3
      memory.available: "500Mi" 4
      nodefs.available: "10%"
      nodefs.inodesFree: "5%"
      imagefs.available: "15%"
      imagefs.inodesFree: "10%"
    evictionSoftGracePeriod: 5
      memory.available: "1m30s"
      nodefs.available: "1m30s"
      nodefs.inodesFree: "1m30s"
      imagefs.available: "1m30s"
```

```
    imagefs.inodesFree: "1m30s"
  evictionHard: 6
    memory.available: "200Mi"
    nodefs.available: "5%"
    nodefs.inodesFree: "4%"
    imagefs.available: "10%"
    imagefs.inodesFree: "5%"
  evictionPressureTransitionPeriod: 0s 7
  imageMinimumGCAge: 5m 8
  imageGCHighThresholdPercent: 80 9
  imageGCLowThresholdPercent: 75 10
```

1. オブジェクトの名前。
2. セレクターラベル。
3. エビクションのタイプ: **evictionSoft** または **evictionHard**。
4. 特定のエビクショントリガーシグナルに基づくエビクションのしきい値。
5. ソフトエビクションの猶予期間。このパラメーターは、**eviction-hard** には適用されません。
6. 特定のエビクショントリガーシグナルに基づくエビクションのしきい値。**evictionHard** の場合、これらのパラメーターをすべて指定する必要があります。すべてのパラメーターを指定しないと、指定したパラメーターのみが適用され、ガベージコレクションが正しく機能しません。
7. エビクション不足の状態から移行するまでの待機時間。
8. ガベージコレクションによって削除されるまでの未使用のイメージの有効期間。
9. イメージのガベージコレクションをトリガーするディスク使用量のパーセント (整数で表される) です。
10. イメージのガベージコレクションが解放しようとするディスク使用量のパーセント (整数で表される) です。

2. オブジェクトを作成します。

```
$ oc create -f <file-name>.yaml
```

以下に例を示します。

```
$ oc create -f gc-container.yaml
```

出力例

```
kubeletconfig.machineconfiguration.openshift.io/gc-container created
```

3. ガベージコレクションがアクティブであることを確認します。カスタムリソースで指定した Machine Config Pool では、変更が完全に実行されるまで **UPDATING** が 'true` と表示されます。

```
$ oc get machineconfigpool
```

出力例

```
NAME     CONFIG                                             UPDATED   UPDATING
master   rendered-master-546383f80705bd5aeaba93   True      False
worker   rendered-worker-b4c51bb33ccaae6fc4a6a5   False     True
```

5.8. OPENSHIFT CONTAINER PLATFORM クラスター内のノードのリソースの割り当て

より信頼性の高いスケジューリングを実現し、ノードにおけるリソースのオーバーコミットを最小限にするために、**kubelet** および **kube-proxy** などの基礎となるノードのコンポーネント、および **sshd** および **NetworkManager** などの残りのシステムコンポーネントに使用される CPU およびメモリーリソースの一部を予約します。予約するリソースを指定して、スケジューラーに、ノードが Pod で使用できる残りの CPU およびメモリーリソースについての詳細を提供します。

5.8.1. ノードにリソースを割り当てる方法について

OpenShift Container Platform 内のノードコンポーネントの予約された CPU とメモリーリソースは、2 つのノード設定に基づいています。

設定	説明
kube-reserved	この設定は OpenShift Container Platform では使用されません。確保する予定の CPU およびメモリーリソースを **system-reserved** 設定に追加します。
system-reserved	この設定は、ノードコンポーネントおよびシステムコンポーネント用に予約するリソースを特定します。デフォルト設定は、OpenShift Container Platform および Machine Config Operator のバージョンによって異なります。**machine-config-operator** リポジトリーでデフォルトの **systemReserved** パラメーターを確認します。

フラグが設定されていない場合、デフォルトが使用されます。いずれのフラグも設定されていない場合、割り当てられるリソースは、割り当て可能なリソースの導入前であるためにノードの容量に設定されます。

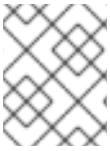
注記

reservedSystemCPUs パラメーターを使用して予約される CPU は、 **kube-reserved** または **system-reserved** を使用した割り当てには使用できません。

5.8.1.1. OpenShift Container Platform による割り当てられたリソースの計算方法

割り当てられたリソースの量は、以下の数式に基づいて計算されます。

```
[Allocatable] = [Node Capacity] - [system-reserved] - [Hard-Eviction-Thresholds]
```



注記

Allocatable の値がノードレベルで Pod に対して適用されるために、**Hard-Eviction-Thresholds** を **Allocatable** から差し引くと、システムの信頼性が強化されます。

Allocatable が負の値の場合、これは **0** に設定されます。

各ノードはコンテナーランタイムおよび kubelet によって利用されるシステムリソースについて報告します。**system-reserved** パラメーターの設定を簡素化するには、ノード要約 API を使用してノードに使用するリソースを表示します。ノードの要約は **/api/v1/nodes/<node>/proxy/stats/summary** で利用できます。

5.8.1.2. ノードによるリソースの制約の適用方法

ノードは、Pod が設定された割り当て可能な値に基づいて消費できるリソースの合計量を制限できます。この機能は、Pod がシステムサービス (コンテナーランタイム、ノードエージェントなど) で必要とされる CPU およびメモリーリソースを使用することを防ぎ、ノードの信頼性を大幅に強化します。ノードの信頼性を強化するために、管理者はリソースの使用についてのターゲットに基づいてリソースを確保する必要があります。

ノードは、QoS (Quality of Service) を適用する新規の cgroup 階層を使用してリソースの制約を適用します。すべての Pod は、システムデーモンから切り離された専用の cgroup 階層で起動されます。

管理者は Guaranteed QoS (Quality of Service) のある Pod と同様にシステムデーモンを処理する必要があります。システムデーモンは、境界となる制御グループ内でバーストする可能性があり、この動作はクラスターのデプロイメントの一部として管理される必要があります。**system-reserved** で CPU およびメモリーリソースの量を指定し、システムデーモンの CPU およびメモリーリソースを予約します。

system-reserved 制限を適用すると、重要なシステムサービスが CPU およびメモリーリソースを受信できなることがあります。その結果、重要なシステムサービスは、out-of-memory killer によって終了する可能性があります。そのため、正確な推定値を判別するためにノードの徹底的なプロファイリングを実行した場合や、そのグループのプロセスが out-of-memory killer によって終了する場合に重要なシステムサービスが確実に復元できる場合にのみ **system-reserved** を適用することが推奨されます。

5.8.1.3. エビクションのしきい値について

ノードがメモリー不足の状態にある場合、ノード全体、およびノードで実行されているすべての Pod に影響が及ぶ可能性があります。たとえば、メモリーの予約量を超える量を使用するシステムデーモンは、メモリー不足のイベントを引き起こす可能性があります。システムのメモリー不足のイベントを防止するか、またはそれが発生する可能性を軽減するために、ノードはリソース不足の処理 (out of resource handling) を行います。

--eviction-hard フラグで一部のメモリーを予約することができます。ノードは、ノードのメモリー可用性が絶対値またはパーセンテージを下回る場合は常に Pod のエビクトを試行します。システムデーモンがノードに存在しない場合、Pod はメモリーの **capacity - eviction-hard** に制限されます。このため、メモリー不足の状態になる前にエビクションのバッファーとして確保されているリソースは Pod で利用することはできません。

以下の例は、割り当て可能なノードのメモリーに対する影響を示しています。

- ノード容量: **32Gi**
- --system-reserved is **3Gi**
- --eviction-hard は **100Mi** に設定される。

このノードについては、有効なノードの割り当て可能な値は **28.9Gi** です。ノードおよびシステムコンポーネントが予約分をすべて使い切る場合、Pod に利用可能なメモリーは **28.9Gi** となり、この使用量を超える場合に kubelet は Pod をエビクトします。

トップレベルの cgroup でノードの割り当て可能分 (**28.9Gi**) を適用する場合、Pod は **28.9Gi** を超えることはできません。エビクションは、システムデーモンが **3.1Gi** よりも多くのメモリーを消費しない限り実行されません。

上記の例ではシステムデーモンが予約分すべてを使い切らない場合も、ノードのエビクションが開始される前に、Pod では境界となる cgroup からの memcg OOM による強制終了が発生します。この状況で QoS をより効果的に実行するには、ノードですべての Pod のトップレベルの cgroup に対し、ハードエビクションしきい値が **Node Allocatable + Eviction Hard Thresholds** になるよう適用できます。

システムデーモンがすべての予約分を使い切らない場合で、Pod が **28.9Gi** を超えるメモリーを消費する場合、ノードは Pod を常にエビクトします。エビクションが時間内に生じない場合には、Pod が **29Gi** のメモリーを消費すると OOM による強制終了が生じます。

5.8.1.4. スケジューラーがリソースの可用性を判別する方法

スケジューラーは、**node.Status.Capacity** ではなく **node.Status.Allocatable** の値を使用して、ノードが Pod スケジューリングの候補になるかどうかを判別します。

デフォルトで、ノードはそのマシン容量をクラスターで完全にスケジュール可能であるとして報告します。

5.8.2. ノードに割り当てられるリソースの設定

OpenShift Container Platform は、割り当てに使用する CPU および メモリーリソースタイプをサポートします。**ephemeral-resource** リソースタイプもサポートされます。**cpu** タイプについては、リソースの数量が、**200m**、**0.5**、または **1** のようにコア単位で指定されます。**memory** および **ephemeral-storage** の場合、**200Ki**、**50Mi**、または **5Gi** などのバイト単位で指定されます。

管理者として、(**cpu=200m,memory=512Mi** などの) **<resource_type>=<resource_quantity>** ペアのセットを使い、カスタムリソース (CR) を使用してこれらを設定することができます。

推奨される **system-reserved** 値の詳細は、推奨される system-reserved 値 を参照してください。

前提条件

1. 設定するノードタイプの静的な **MachineConfigPool** CRD に関連付けられたラベルを取得します。以下のいずれかの手順を実行します。

   a. Machine Config Pool を表示します。

   ```
   $ oc describe machineconfigpool <name>
   ```

   以下に例を示します。

   ```
   $ oc describe machineconfigpool worker
   ```

   出力例

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: MachineConfigPool
   metadata:
   ```

```
  creationTimestamp: 2019-02-08T14:52:39Z
  generation: 1
  labels:
    custom-kubelet: small-pods (1)
```

(1) ラベルが追加されると、**labels** の下に表示されます。

b. ラベルが存在しない場合は、キー/値のペアを追加します。

```
$ oc label machineconfigpool worker custom-kubelet=small-pods
```

手順

1. 設定変更のためのカスタムリソース (CR) を作成します。

リソース割り当て CR の設定例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: set-allocatable (1)
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: small-pods (2)
  kubeletConfig:
    systemReserved:
      cpu: 1000m
      memory: 1Gi
```

(1) CR に名前を割り当てます。

(2) Machine Config Pool からラベルを指定します。

5.9. クラスター内のノードの特定 CPU の割り当て

静的 CPU マネージャーポリシー を使用する場合、クラスター内の特定のノードで使用できるように特定の CPU を予約できます。たとえば、24 CPU のあるシステムでは、コントロールプレーン用に 0 - 3 の番号が付けられた CPU を予約して、コンピュートノードが CPU 4 - 23 を使用できるようにすることができます。

5.9.1. ノードの CPU の予約

特定のノード用に予約される CPU の一覧を明示的に定義するには、**KubeletConfig** カスタムリソース (CR) を作成して **reservedSystemCPUs** パラメーターを定義します。この一覧は、**systemReserved** および **kubeReserved** パラメーターを使用して予約される可能性のある CPU に対して優先されます。

手順

1. 設定する必要のあるノードタイプの Machine Config Pool (MCP) に関連付けられたラベルを取得します。

-

```
$ oc describe machineconfigpool <name>
```

以下に例を示します。

```
$ oc describe machineconfigpool worker
```

出力例

```
Name:         worker
Namespace:
Labels:       machineconfiguration.openshift.io/mco-built-in=
              pools.operator.machineconfiguration.openshift.io/worker= 1
Annotations:  <none>
API Version:  machineconfiguration.openshift.io/v1
Kind:         MachineConfigPool
...
```

1. MCP ラベルを取得します。

2. **KubeletConfig** CR の YAML ファイルを作成します。

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: set-reserved-cpus 1
spec:
  kubeletConfig:
    reservedSystemCPUs: "0,1,2,3" 2
  machineConfigPoolSelector:
    matchLabels:
      pools.operator.machineconfiguration.openshift.io/worker: "" 3
```

1. CR の名前を指定します。
2. MCP に関連付けられたノード用に予約する CPU のコア ID を指定します。
3. MCP からラベルを指定します。

3. CR オブジェクトを作成します。

```
$ oc create -f <file_name>.yaml
```

関連情報

- **systemReserved** および **kubeReserved** パラメーターについての詳細は、OpenShift Container Platform クラスターのノードのリソース割り当て について参照してください。

5.10. MACHINE CONFIG DAEMON メトリクス

Machine Config Daemon は Machine Config Operator の一部です。これはクラスター内のすべてのノードで実行されます。Machine Config Daemon は、各ノードの設定変更および更新を管理します。

5.10.1. Machine Config Daemon メトリクス

OpenShift Container Platform 4.3 以降、Machine Config Daemon はメトリクスのセットを提供します。これらのメトリクスには、Prometheus クラスターモニターリングスタックを使用してアクセスできます。

以下の表では、これらのメトリクスのセットについて説明しています。

注記

*Name*列と**Description**列に * が付いているメトリックは、パフォーマンスの問題を引き起こす可能性のある重大なエラーを表します。このような問題により、更新およびアップグレードが続行されなくなる可能性があります。

注記

一部のエントリーには特定のログを取得するコマンドが含まれていますが、最も包括的なログのセットは、**oc adm must-gather** コマンドを使用して利用できます。

表5.4 MCO メトリクス

名前	フォーマット	説明	備考
mcd_host_os_and_version	**[]string{"os", "version"}**	RHCOS や RHEL など、MCD が実行されている OS を示します。RHCOS の場合、バージョンは指定されます。	
ssh_accessed	**counter**	ノードへの SSH 認証に成功した数を表示します。	ゼロ以外の値は、いずれかのユーザーがノードに手動で変更した可能性があることを示しています。このような変更により、ディスクの状態とマシン設定で定義される状態の差異により、調整不可能なエラーが発生する可能性があります。

名前	フォーマット	説明	備考
mcd_drain*	**{"drain_time", "err"}**	ドレイン (解放) の失敗時に受信されるエラーをログに記録します。*	ドレイン (解放) が成功するには、複数回試行する必要がある可能性があり、ターミナルでは、ドレイン (解放) に失敗すると更新を続行できなくなります。ドレイン (解放) にかかる時間を示す **drain_time** メトリクスはトラブルシューティングに役立つ可能性があります。 詳細な調査を実行するには、以下を実行してログを表示します。 **$ oc logs -f -n openshift-machine-config-operator machine-config-daemon-<hash> -c machine-config-daemon**
mcd_pivot_err*	**[]string{"pivot_target", "err"}**	ピボットで発生するログ。*	ピボットのエラーにより、OS のアップグレードを続行できなくなる可能性があります。 詳細な調査を行うには、以下のコマンドを実行してノードにアクセスし、そのすべてのログを表示します。 **$ oc debug node/<node> — chroot /host journalctl -u pivot.service** または、以下のコマンドを実行して、**machine-config-daemon** コンテナーのログのみを確認します。 **$ oc logs -f -n openshift-machine-config-operator machine-config-daemon-<hash> -c machine-config-daemon**
mcd_state	**[]string{"state", "reason"}**	指定ノードの Machine Config Daemon の状態。状態のオプションとして、Done、Working、および Degraded があります。Degraded の場合は、理由も含まれます。	詳細な調査を実行するには、以下を実行してログを表示します。 **$ oc logs -f -n openshift-machine-config-operator machine-config-daemon-<hash> -c machine-config-daemon**

名前	フォーマット	説明	備考
mcd_kubelet_state*	**[]string{"err"}**	kubelet の正常性についての失敗をログに記録します。*	これは、失敗数が 0 で空になることが予想されます。失敗数が 2 を超えると、しきい値を超えたことを示すエラーが出されます。これは kubelet の正常性に関連した問題の可能性を示します。 詳細な調査を行うには、以下のコマンドを実行してノードにアクセスし、そのすべてのログを表示します。 **$ oc debug node/<node> — chroot /host journalctl -u kubelet**
mcd_reboot_err*	**[]string{"message", "err"}**	再起動の失敗と対応するエラーをログに記録します。*	これは空になることが予想されますが、これは再起動が成功したことを示します。 詳細な調査を実行するには、以下を実行してログを表示します。 **$ oc logs -f -n openshift-machine-config-operator machine-config-daemon-<hash> -c machine-config-daemon**
mcd_update_state	**[]string{"config", "err"}**	設定更新の成功または失敗、および対応するエラーをログに記録します。	予想される値は **rendered-master/rendered-worker-XXXX** です。更新に失敗すると、エラーが表示されます。 詳細な調査を実行するには、以下を実行してログを表示します。 **$ oc logs -f -n openshift-machine-config-operator machine-config-daemon-<hash> -c machine-config-daemon**

関連情報

- モニターリングの概要 を参照してください。
- クラスターに関するデータの収集についてのドキュメント を参照してください。

第6章 コンテナーの使用

6.1. コンテナーについて

OpenShift Container Platform アプリケーションの基本的な単位は **コンテナー** と呼ばれています。Linux コンテナーテクノロジー は、指定されたリソースのみと対話するために実行中のプロセスを分離する軽量なメカニズムです。

数多くのアプリケーションインスタンスは、相互のプロセス、ファイル、ネットワークなどを可視化せずに単一ホストのコンテナーで実行される可能性があります。通常、コンテナーは任意のワークロードに使用されますが、各コンテナーは Web サーバーまたはデータベースなどの (通常はマイクロサービスと呼ばれることの多い) 単一サービスを提供します。

Linux カーネルは数年にわたりコンテナーテクノロジーの各種機能を統合してきました。OpenShift Container Platform および Kubernetes は複数ホストのインストール間でコンテナーのオーケストレーションを実行する機能を追加します。

コンテナーおよび RHEL カーネルメモリーについて
Red Hat Enterprise Linux (RHEL) の動作により、CPU 使用率の高いノードのコンテナーは、予想以上に多いメモリーを消費しているように見える可能性があります。メモリー消費量の増加は、RHEL カーネルの **kmem_cache** によって引き起こされる可能性があります。RHEL カーネルは、それぞれの cgroup に **kmem_cache** を作成します。パフォーマンスの強化のために、**kmem_cache** には **cpu_cache** と任意の NUMA ノードのノードキャッシュが含まれます。これらのキャッシュはすべてカーネルメモリーを消費します。

これらのキャッシュに保存されるメモリーの量は、システムが使用する CPU の数に比例します。結果として、CPU の数が増えると、より多くのカーネルメモリーがこれらのキャッシュに保持されます。これらのキャッシュのカーネルメモリーの量が増えると、OpenShift Container Platform コンテナーで設定済みのメモリー制限を超える可能性があり、これにより、コンテナーが強制終了される可能性があります。

カーネルメモリーの問題によりコンテナーが失われないようにするには、コンテナーが十分なメモリーを要求することを確認します。以下の式を使用して、**kmem_cache** が消費するメモリー量を見積ることができます。この場合、**nproc** は、**nproc** コマンドで報告される利用可能なプロセス数です。コンテナーの要求の上限が低くなる場合、この値にコンテナーメモリーの要件を加えた分になります。

```
$(nproc) X 1/2 MiB
```

6.2. POD のデプロイ前の、INIT コンテナーの使用によるタスクの実行

OpenShift Container Platform は、**Init コンテナー** を提供します。 このコンテナーは、アプリケーションコンテナーの前に実行される特殊なコンテナーであり、アプリのイメージに存在しないユーティリティーまたはセットアップスクリプトを含めることができます。

6.2.1. Init コンテナーについて

Pod の残りの部分がデプロイされる前に、init コンテナーリソースを使用して、タスクを実行することができます。

Pod は、アプリケーションコンテナーに加えて、init コンテナーを持つことができます。Init コンテナーにより、セットアップスクリプトとバインディングコードを再編成できます。

init コンテナーは以下のことを行うことができます。

- セキュリティー上の理由のためにアプリケーションコンテナーイメージに含めることが望ましくないユーティリティーを含めることができ、それらを実行できます。
- アプリのイメージに存在しないセットアップに必要なユーティリティーまたはカスタムコードを含めることができます。たとえば、単に Sed、Awk、Python、Dig のようなツールをセットアップ時に使用するために別のイメージからイメージを作成する必要はありません。
- Linux namespace を使用して、アプリケーションコンテナーがアクセスできないシークレットへのアクセスなど、アプリケーションコンテナーとは異なるファイルシステムビューを設定できます。

各 init コンテナーは、次のコンテナーが起動する前に正常に完了している必要があります。そのため、Init コンテナーには、一連の前提条件が満たされるまでアプリケーションコンテナーの起動をブロックしたり、遅延させたりする簡単な方法となります。

たとえば、以下は init コンテナーを使用するいくつかの方法になります。

- 以下のようなシェルコマンドでサービスが作成されるまで待機します。

```
for i in {1..100}; do sleep 1; if dig myservice; then exit 0; fi; done; exit 1
```

- 以下のようなコマンドを使用して、Downward API からリモートサーバーにこの Pod を登録します。

```
$ curl -X POST
http://$MANAGEMENT_SERVICE_HOST:$MANAGEMENT_SERVICE_PORT/register -d
‘instance=$()&ip=$()’
```

- **sleep 60** のようなコマンドを使用して、アプリケーションコンテナーが起動するまでしばらく待機します。
- Git リポジトリーのクローンをボリュームに作成します。
- 設定ファイルに値を入力し、テンプレートツールを実行して、主要なアプリコンテナーの設定ファイルを動的に生成します。たとえば、設定ファイルに POD_IP の値を入力し、Jinja を使用して主要なアプリ設定ファイルを生成します。

詳細は、Kubernetes ドキュメント を参照してください。

6.2.2. Init コンテナーの作成

以下の例は、2 つの init コンテナーを持つ単純な Pod の概要を示しています。1 つ目は **myservice** を待機し、2 つ目は **mydb** を待機します。両方のコンテナーが完了すると、Pod が開始されます。

手順

1. init コンテナーの YAML ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
```

```
  containers:
  - name: myapp-container
    image: registry.access.redhat.com/ubi8/ubi:latest
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: registry.access.redhat.com/ubi8/ubi:latest
    command: ['sh', '-c', 'until getent hosts myservice; do echo waiting for myservice; sleep 2;
done;']
  - name: init-mydb
    image: registry.access.redhat.com/ubi8/ubi:latest
    command: ['sh', '-c', 'until getent hosts mydb; do echo waiting for mydb; sleep 2; done;']
```

2. **myservice** サービス用の YAML ファイルを作成します。

```
kind: Service
apiVersion: v1
metadata:
  name: myservice
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

3. **mydb** サービス用の YAML ファイルを作成します。

```
kind: Service
apiVersion: v1
metadata:
  name: mydb
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9377
```

4. 以下のコマンドを実行して **myapp-pod** を作成します。

```
$ oc create -f myapp.yaml
```

出力例

```
pod/myapp-pod created
```

5. Pod のステータスを表示します。

```
$ oc get pods
```

出力例

```
NAME                  READY     STATUS            RESTARTS   AGE
myapp-pod             0/1       Init:0/2          0          5s
```

Pod のステータスが、待機状態であることを示していることを確認します。

6. 以下のコマンドを実行してサービスを作成します。

```
$ oc create -f mydb.yaml
```

```
$ oc create -f myservice.yaml
```

7. Pod のステータスを表示します。

```
$ oc get pods
```

出力例

```
NAME                     READY     STATUS            RESTARTS   AGE
myapp-pod                1/1       Running           0          2m
```

6.3. ボリュームの使用によるコンテナーデータの永続化

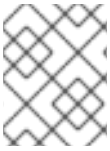
コンテナー内のファイルは一時的なものです。そのため、コンテナーがクラッシュしたり停止したりした場合は、データが失われます。**ボリューム** を使用すると、Pod 内のコンテナーが使用しているデータを永続化できます。ボリュームはディレクトリーであり、Pod 内のコンテナーからアクセスすることができます。 ここでは、データが Pod の有効期間中保存されます。

6.3.1. ボリュームについて

ボリュームとは Pod およびコンテナーで利用可能なマウントされたファイルシステムのことであり、これらは数多くのホストのローカルまたはネットワーク割り当てストレージのエンドポイントでサポートされる場合があります。コンテナーはデフォルトで永続性がある訳ではなく、それらのコンテンツは再起動時にクリアされます。

ボリュームのファイルシステムにエラーが含まれないようにし、かつエラーが存在する場合はそれを修復するために、OpenShift Container Platform は **mount** ユーティリティーの前に **fsck** ユーティリティーを起動します。これはボリュームを追加するか、または既存ボリュームを更新する際に実行されます。

最も単純なボリュームタイプは **emptyDir** です。これは、単一マシンの一時的なディレクトリーです。管理者はユーザーによる Pod に自動的に割り当てられる 永続ボリュームの要求を許可することもできます。



注記

emptyDir ボリュームストレージは、FSGroup パラメーターがクラスター管理者によって有効にされている場合は Pod の FSGroup に基づいてクォータで制限できます。

6.3.2. OpenShift Container Platform CLI によるボリュームの操作

CLI コマンド **oc set volume** を使用して、レプリケーションコントローラーやデプロイメント設定などの Pod テンプレートを持つオブジェクトのボリュームおよびボリュームマウントを追加し、削除することができます。また、Pod または Pod テンプレートを持つオブジェクトのボリュームを一覧表示することもできます。

oc set volume コマンドは以下の一般的な構文を使用します。

```
$ oc set volume <object_selection> <operation> <mandatory_parameters> <options>
```

オブジェクトの選択

oc set volume コマンドの **object_seletion** パラメーターに、以下のいずれかを指定します。

表6.1 オブジェクトの選択

構文	説明	例
<object_type> <name>	タイプ **<object_type>** の **<name>** を選択します。	**deploymentConfig registry**
<object_type>/<name>	タイプ **<object_type>** の **<name>** を選択します。	**deploymentConfig/registry**
<object_type> --selector=<object_label_selector>	所定のラベルセレクターに一致するタイプ **<object_type>** のリソースを選択します。	**deploymentConfig --selector="name=registry"**
<object_type> --all	タイプ **<object_type>** のすべてのリソースを選択します。	**deploymentConfig --all**
-f または **--filename=<file_name>**	リソースを編集するために使用するファイル名、ディレクトリー、または URL です。	**-f registry-deployment-config.json**

操作

oc set volume コマンドの **operation** パラメーターに **--add** または **--remove** を指定します。

必須パラメーター

いずれの必須パラメーターも選択された操作に固有のものであり、これらについては後のセクションで説明します。

オプション

いずれのオプションも選択された操作に固有のものであり、これらについては後のセクションで説明します。

6.3.3. Pod のボリュームとボリュームマウントの一覧表示

Pod または Pod テンプレートのボリュームおよびボリュームマウントを一覧表示することができます。

手順

ボリュームを一覧表示するには、以下の手順を実行します。

```
$ oc set volume <object_type>/<name> [options]
```

ボリュームのサポートされているオプションを一覧表示します。

オプション	説明	デフォルト
--name	ボリュームの名前。	
-c, --containers	名前でコンテナーを選択します。すべての文字に一致するワイルドカード **'*'** を取ることもできます。	**'*'**

以下に例を示します。

- Pod **p1** のすべてのボリュームを一覧表示するには、以下を実行します。

```
$ oc set volume pod/p1
```

- すべてのデプロイメント設定で定義されるボリューム **v1** を一覧表示するには、以下の手順を実行します。

```
$ oc set volume dc --all --name=v1
```

6.3.4. Pod へのボリュームの追加

Pod にボリュームとボリュームマウントを追加することができます。

手順

ボリューム、ボリュームマウントまたはそれらの両方を Pod テンプレートに追加するには、以下を実行します。

```
$ oc set volume <object_type>/<name> --add [options]
```

表6.2 ボリュームを追加するためのサポートされるオプション

オプション	説明	デフォルト
--name	ボリュームの名前。	指定がない場合は、自動的に生成されます。
-t, --type	ボリュームソースの名前。サポートされる値は **emptyDir**、**hostPath**、**secret**、**configmap**、**persistentVolumeClaim** または **projected** です。	**emptyDir**
-c, --containers	名前でコンテナーを選択します。すべての文字に一致するワイルドカード **'*'** を取ることもできます。	**'*'**

オプション	説明	デフォルト
-m, --mount-path	選択されたコンテナー内のマウントパス。コンテナーのルート (/) や、ホストとコンテナーで同じパスにはマウントしないでください。これは、コンテナーに十分な特権が付与されている場合、ホストシステムを破壊する可能性があります (例: ホストの **/dev/pts** ファイル)。ホストをマウントするには、**/host** を使用するのが安全です。	
--path	ホストパス。**--type=hostPath** の必須パラメーターです。コンテナーのルート (/) や、ホストとコンテナーで同じパスにはマウントしないでください。これは、コンテナーに十分な特権が付与されている場合、ホストシステムを破壊する可能性があります (例: ホストの **/dev/pts** ファイル)。ホストをマウントするには、**/host** を使用するのが安全です。	
--secret-name	シークレットの名前。**--type=secret** の必須パラメーターです。	
--configmap-name	configmap の名前。**--type=configmap** の必須のパラメーターです。	
--claim-name	永続ボリューム要求 (PVC) の名前。**--type=persistentVolumeClaim** の必須パラメーターです。	
--source	JSON 文字列としてのボリュームソースの詳細。必要なボリュームソースが **--type** でサポートされない場合に推奨されます。	
-o, --output	サーバー上で更新せずに変更したオブジェクトを表示します。サポートされる値は **json**、**yaml** です。	

オプション	説明	デフォルト
--output-version	指定されたバージョンで変更されたオブジェクトを出力します。	**api-version**

以下に例を示します。

- 新規ボリュームソース **emptyDir** を **registry DeploymentConfig** オブジェクトに追加するには、以下を実行します。

```
$ oc set volume dc/registry --add
```

- レプリケーションコントローラー **r1** のシークレット **secret1** を使用してボリューム **v1** を追加し、コンテナー内の **/data** でマウントするには、以下を実行します。

```
$ oc set volume rc/r1 --add --name=v1 --type=secret --secret-name='secret1' --mount-
path=/data
```

- 要求名 **pvc1** を使って既存の永続ボリューム **v1** をディスク上のデプロイメント設定 **dc.json** に追加し、ボリュームをコンテナー **c1** の **/data** にマウントし、サーバー上で **DeploymentConfig** オブジェクトを更新します。

```
$ oc set volume -f dc.json --add --name=v1 --type=persistentVolumeClaim \
  --claim-name=pvc1 --mount-path=/data --containers=c1
```

- すべてのレプリケーションコントローラー向けにリビジョン **5125c45f9f563** を使い、Git リポジトリー **https://github.com/namespace1/project1** に基づいてボリューム **v1** を追加するには、以下の手順を実行します。

```
$ oc set volume rc --all --add --name=v1 \
  --source='{"gitRepo": {
              "repository": "https://github.com/namespace1/project1",
              "revision": "5125c45f9f563"
            }}'
```

6.3.5. Pod 内のボリュームとボリュームマウントの更新

Pod 内のボリュームとボリュームマウントを変更することができます。

手順

--overwrite オプションを使用して、既存のボリュームを更新します。

```
$ oc set volume <object_type>/<name> --add --overwrite [options]
```

以下に例を示します。

- レプリケーションコントローラー **r1** の既存ボリューム **v1** を既存の永続ボリューム要求 (PVC) **pvc1** に置き換えるには、以下の手順を実行します。

```
$ oc set volume rc/r1 --add --overwrite --name=v1 --type=persistentVolumeClaim --claim-
name=pvc1
```

- **DeploymentConfig** オブジェクトの **d1** のマウントポイントを、ボリューム **v1**の **/opt** に変更するには、以下を実行します。

```
$ oc set volume dc/d1 --add --overwrite --name=v1 --mount-path=/opt
```

6.3.6. Pod からのボリュームおよびボリュームマウントの削除

Pod からボリュームまたはボリュームマウントを削除することができます。

手順

Pod テンプレートからボリュームを削除するには、以下を実行します。

```
$ oc set volume <object_type>/<name> --remove [options]
```

表6.3 ボリュームを削除するためにサポートされるオプション

オプション	説明	デフォルト
--name	ボリュームの名前。	
-c, --containers	名前でコンテナーを選択します。すべての文字に一致するワイルドカード **'*'** を取ることもできます。	**'*'**
--confirm	複数のボリュームを 1 度に削除することを示します。	
-o, --output	サーバー上で更新せずに変更したオブジェクトを表示します。サポートされる値は **json**、**yaml** です。	
--output-version	指定されたバージョンで変更されたオブジェクトを出力します。	**api-version**

以下に例を示します。

- **DeploymentConfig** オブジェクトの **d1**から ボリューム **v1** を削除するには、以下を実行します。

```
$ oc set volume dc/d1 --remove --name=v1
```

- **DeploymentConfig** オブジェクトの **d1** の **c1** のコンテナーからボリューム **v1** をアンマウントし、**d1** のコンテナーで参照されていない場合にボリューム **v1** を削除するには、以下の手順を実行します。

```
$ oc set volume dc/d1 --remove --name=v1 --containers=c1
```

- レプリケーションコントローラー **r1** のすべてのボリュームを削除するには、以下の手順を実行します。

```
$ oc set volume rc/r1 --remove --confirm
```

6.3.7. Pod 内での複数の用途のためのボリュームの設定

ボリュームを、単一 Pod で複数の使用目的のためにボリュームを共有するように設定できます。この場合、**volumeMounts.subPath** プロパティーを使用し、ボリュームのルートの代わりにボリューム内に **subPath** 値を指定します。

手順

1. ボリューム内のファイルの一覧を表示して、**oc rsh** コマンドを実行します。

```
$ oc rsh <pod>
```

出力例

```
sh-4.2$ ls /path/to/volume/subpath/mount
example_file1 example_file2 example_file3
```

2. **subPath** を指定します。

subPath パラメーターを含む Pod 仕様の例

```
apiVersion: v1
kind: Pod
metadata:
  name: my-site
spec:
    containers:
    - name: mysql
      image: mysql
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: site-data
        subPath: mysql 1
    - name: php
      image: php
      volumeMounts:
      - mountPath: /var/www/html
        name: site-data
        subPath: html 2
    volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-site-data
```

1 データベースは **mysql** フォルダーに保存されます。

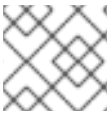
2 HTML コンテンツは **html** フォルダーに保存されます。

6.4. PROJECTED ボリュームによるボリュームのマッピング

Projected ボリューム は、いくつかの既存のボリュームソースを同じディレクトリーにマップします。

以下のタイプのボリュームソースを展開できます。

- シークレット
- Config Map
- Downward API



注記

すべてのソースは Pod と同じ namespace に置かれる必要があります。

6.4.1. Projected ボリュームについて

Projected ボリュームはこれらのボリュームソースの任意の組み合わせを単一ディレクトリーにマップし、ユーザーの以下の実行を可能にします。

- 単一ボリュームを、複数のシークレットのキー、設定マップ、および Downward API 情報で自動的に設定し、各種の情報ソースで単一ディレクトリーを合成できるようにします。
- 各項目のパスを明示的に指定して、単一ボリュームを複数シークレットのキー、設定マップ、および Downward API 情報で設定し、ユーザーがボリュームの内容を完全に制御できるようにします。

重要

RunAsUser パーミッションが Linux ベースの Pod のセキュリティーコンテキストに設定されている場合、Projected ファイルには、コンテナーユーザー所有権を含む適切なパーミッションが設定されます。ただし、Windows の同等の **RunAsUsername** パーミッションが Windows Pod に設定されている場合、kubelet は Projected ボリュームのファイルに正しい所有権を設定できません。

そのため、Windows Pod のセキュリティーコンテキストに設定された **RunAsUsername** パーミッションは、OpenShift Container Platform で実行される Windows の Projected ボリュームには適用されません。

以下の一般的なシナリオは、Projected ボリュームを使用する方法について示しています。

設定マップ、シークレット、Downward API

Projected ボリュームを使用すると、パスワードが含まれる設定データでコンテナーをデプロイできます。これらのリソースを使用するアプリケーションは、Red Hat OpenStack Platform (RHOSP) を Kubernetes にデプロイしている可能性があります。設定データは、サービスが実稼働用またはテストで使用されるかによって異なった方法でアセンブルされる必要がある可能性があります。Pod に実稼働またはテストのラベルが付けられている場合、Downward API セレクター **metadata.labels** を使用して適切な RHOSP 設定を生成できます。

設定マップ + シークレット

Projected ボリュームにより、設定データおよびパスワードを使用してコンテナーをデプロイできます。たとえば、設定マップを、Vault パスワードファイルを使用して暗号解除する暗号化された機密タスクで実行する場合があります。

ConfigMap + Downward API.

Projected ボリュームにより、Pod 名 (**metadata.name** セレクターで選択可能) を含む設定を生成できます。このアプリケーションは IP トラッキングを使用せずに簡単にソースを判別できるよう要求と共に Pod 名を渡すことができます。

シークレット + Downward API

Projected ボリュームにより、Pod の namespace (**metadata.namespace** セレクターで選択可能) を暗号化するためのパブリックキーとしてシークレットを使用できます。この例では、Operator はこのアプリケーションを使用し、暗号化されたトランスポートを使用せずに namespace 情報を安全に送信できるようになります。

6.4.1.1. Pod 仕様の例

以下は、Projected ボリュームを作成するための **Pod** 仕様の例です。

シークレット、Downward API および設定マップを含む Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts: 1
    - name: all-in-one
      mountPath: "/projected-volume" 2
      readOnly: true 3
  volumes: 4
  - name: all-in-one 5
    projected:
      defaultMode: 0400 6
      sources:
      - secret:
          name: mysecret 7
          items:
            - key: username
              path: my-group/my-username 8
      - downwardAPI: 9
          items:
            - path: "labels"
              fieldRef:
                fieldPath: metadata.labels
            - path: "cpu_limit"
              resourceFieldRef:
                containerName: container-test
                resource: limits.cpu
      - configMap: 10
          name: myconfigmap
          items:
            - key: config
              path: my-group/my-config
              mode: 0777 11
```

1. シークレットを必要とする各コンテナーの **volumeMounts** セクションを追加します。
2. シークレットが表示される未使用ディレクトリーのパスを指定します。
3. **readOnly** を **true** に設定します。
4. それぞれの Projected ボリュームソースを一覧表示するために **volumes** ブロックを追加します。
5. ボリュームの名前を指定します。
6. ファイルに実行パーミッションを設定します。
7. シークレットを追加します。シークレットオブジェクトの名前を追加します。使用する必要のあるそれぞれのシークレットは一覧表示される必要があります。
8. **mountPath** の下にシークレットへのパスを指定します。ここで、シークレットファイルは **/projected-volume/my-group/my-username** になります。
9. Downward API ソースを追加します。
10. ConfigMap ソースを追加します。
11. 特定の展開におけるモードを設定します。

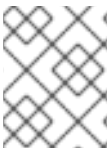
注記

Pod に複数のコンテナーがある場合、それぞれのコンテナーには **volumeMounts** セクションが必要ですが、1 つの **volumes** セクションのみが必要になります。

デフォルト以外のパーミッションモデルが設定された複数シークレットを含む Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      defaultMode: 0755
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/my-username
      - secret:
          name: mysecret2
```

```
    items:
      - key: password
        path: my-group/my-password
        mode: 511
```



注記

defaultMode は展開されるレベルでのみ指定でき、各ボリュームソースには指定されません。ただし、上記のように個々の展開についての **mode** を明示的に指定できます。

6.4.1.2. パスについての留意事項

設定されるパスが同一である場合のキー間の競合

複数のキーを同じパスで設定する場合、Pod 仕様は有効な仕様として受け入れられません。以下の例では、**mysecret** および **myconfigmap** に指定されるパスは同じです。

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/data
      - configMap:
          name: myconfigmap
          items:
            - key: config
              path: my-group/data
```

ボリュームファイルのパスに関連する以下の状況を検討しましょう。

設定されたパスのないキー間の競合

上記のシナリオの場合と同様に、実行時の検証が実行される唯一のタイミングはすべてのパスが Pod の作成時に認識される時です。それ以外の場合は、競合の発生時に指定された最新のリソースがこれより前のすべてのものを上書きします (これは Pod 作成後に更新されるリソースについても同様です)。

1つのパスが明示的なパスであり、もう1つのパスが自動的に展開されるパスである場合の競合

自動的に展開されるデータに一致するユーザー指定パスによって競合が生じる場合、前述のように後からのリソースがこれより前のすべてのものを上書きします。

6.4.2. Pod の Projected ボリュームの設定

Projected ボリュームを作成する場合は、**Pprojected ボリュームについて**で説明されているボリュームファイルパスの状態を考慮します。

以下の例では、Projected ボリュームを使用して、既存のシークレットボリュームソースをマウントする方法が示されています。以下の手順は、ローカルファイルからユーザー名およびパスワードのシークレットを作成するために実行できます。その後に、シークレットを同じ共有ディレクトリーにマウントするために Projected ボリュームを使用して 1 つのコンテナーを実行する Pod を作成します。

手順

既存のシークレットボリュームソースをマウントするために Projected ボリュームを使用するには、以下を実行します。

1. 以下を入力し、パスワードおよびユーザー情報を適宜置き換えて、シークレットを含むファイルを作成します。

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  pass: MWYyZDFlMmU2N2Rm
  user: YWRtaW4=
```

user および **pass** の値には、base64 でエンコーティングされた任意の有効な文字列を使用できます。

以下の例は、base64 の **admin** を示しています。

```
$ echo -n "admin" | base64
```

出力例

```
YWRtaW4=
```

以下の例は、base64 のパスワード **1f2d1e2e67df** を示しています。

```
$ echo -n "1f2d1e2e67df" | base64
```

出力例

```
MWYyZDFlMmU2N2Rm
```

2. 以下のコマンドを使用してシークレットを作成します。

```
$ oc create -f <secrets-filename>
```

以下に例を示します。

```
$ oc create -f secret.yaml
```

出力例

```
secret "mysecret" created
```

3. シークレットが以下のコマンドを使用して作成されていることを確認できます。

```
$ oc get secret <secret-name>
```

以下に例を示します。

```
$ oc get secret mysecret
```

出力例

```
NAME       TYPE      DATA      AGE
mysecret   Opaque    2         17h
```

```
$ oc get secret <secret-name> -o yaml
```

以下に例を示します。

```
$ oc get secret mysecret -o yaml
```

```
apiVersion: v1
data:
  pass: MWYyZDFlMmU2N2Rm
  user: YWRtaW4=
kind: Secret
metadata:
  creationTimestamp: 2017-05-30T20:21:38Z
  name: mysecret
  namespace: default
  resourceVersion: "2107"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: 959e0424-4575-11e7-9f97-fa163e4bd54c
type: Opaque
```

4. **volumes** セクションが含まれる以下のような Pod 設定ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: test-projected-volume
spec:
  containers:
  - name: test-projected-volume
    image: busybox
    args:
    - sleep
    - "86400"
    volumeMounts:
    - name: all-in-one
```

```
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:      (1)
          name: user
      - secret:      (2)
          name: pass
```

(1) (2) 作成されたシークレットの名前。

5. 設定ファイルから Pod を作成します。

```
$ oc create -f <your_yaml_file>.yaml
```

以下に例を示します。

```
$ oc create -f secret-pod.yaml
```

出力例

```
pod "test-projected-volume" created
```

6. Pod コンテナーが実行中であることを確認してから、Pod への変更を確認します。

```
$ oc get pod <name>
```

以下に例を示します。

```
$ oc get pod test-projected-volume
```

出力は以下のようになります。

出力例

```
NAME                    READY     STATUS    RESTARTS   AGE
test-projected-volume   1/1       Running   0          14s
```

7. 別のターミナルで、**oc exec** コマンドを使用し、実行中のコンテナーに対してシェルを開きます。

```
$ oc exec -it <pod> <command>
```

以下に例を示します。

```
$ oc exec -it test-projected-volume -- /bin/sh
```

8. シェルで、**projected-volumes** ディレクトリーに展開されるソースが含まれることを確認します。

```
/ # ls
```

出力例

```
bin            home           root           tmp
dev            proc           run            usr
etc            projected-volume  sys         var
```

6.5. コンテナーによる API オブジェクト使用の許可

Downward API は、OpenShift Container Platform に結合せずにコンテナーが API オブジェクトについての情報を使用できるメカニズムです。この情報には、Pod の名前、namespace およびリソース値が含まれます。コンテナーは、環境変数やボリュームプラグインを使用して Downward API からの情報を使用できます。

6.5.1. Downward API の使用によるコンテナーへの Pod 情報の公開

Downward API には、Pod の名前、プロジェクト、リソースの値などの情報が含まれます。コンテナーは、環境変数やボリュームプラグインを使用して Downward API からの情報を使用できます。

Pod 内のフィールドは、**FieldRef** API タイプを使用して選択されます。**FieldRef** には 2 つのフィールドがあります。

フィールド	説明
fieldPath	Pod に関連して選択するフィールドのパスです。
apiVersion	**fieldPath** セレクターの解釈に使用する API バージョンです。

現時点で v1 API の有効なセレクターには以下が含まれます。

セレクター	説明
metadata.name	Pod の名前です。これは環境変数およびボリュームでサポートされています。
metadata.namespace	Pod の namespace です。 これは環境変数およびボリュームでサポートされています。
metadata.labels	Pod のラベルです。これはボリュームでのみサポートされ、環境変数ではサポートされていません。
metadata.annotations	Pod のアノテーションです。これはボリュームでのみサポートされ、環境変数ではサポートされていません。
status.podIP	Pod の IP です。これは環境変数でのみサポートされ、ボリュームではサポートされていません。

apiVersion フィールドは、指定されていない場合は、対象の Pod テンプレートの API バージョンにデフォルト設定されます。

6.5.2. Downward API を使用してコンテナーの値を使用する方法について

コンテナーは、環境変数やボリュームプラグインを使用して API の値を使用することができます。選択する方法により、コンテナーは以下を使用できます。

- Pod の名前
- Pod プロジェクト/namespace
- Pod のアノテーション
- Pod のラベル

アノテーションとラベルは、ボリュームプラグインのみを使用して利用できます。

6.5.2.1. 環境変数の使用によるコンテナー値の使用

コンテナーの環境変数を設定する際に、**EnvVar** タイプの **valueFrom** フィールド (タイプは **EnvVarSource**) を使用して、変数の値が **value** フィールドで指定されるリテラル値ではなく、**FieldRef** ソースからの値になるように指定します。

この方法で使用できるのは Pod の定数属性のみです。変数の値の変更についてプロセスに通知する方法でプロセスを起動すると、環境変数を更新できなくなるためです。環境変数を使用してサポートされるフィールドには、以下が含まれます。

- Pod の名前
- Pod プロジェクト/namespace

手順

環境変数を使用するには、以下を実行します。

1. **pod.yaml** ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: MY_POD_NAMESPACE
          valueFrom:
```

```
          fieldRef:
            fieldPath: metadata.namespace
  restartPolicy: Never
```

2. **pod.yaml** ファイルから Pod を作成します。

```
$ oc create -f pod.yaml
```

3. コンテナーのログで **MY_POD_NAME** および **MY_POD_NAMESPACE** の値を確認します。

```
$ oc logs -p dapi-env-test-pod
```

6.5.2.2. ボリュームプラグインを使用したコンテナー値の使用

コンテナーは、ボリュームプラグイン使用して API 値を使用できます。

コンテナーは、以下を使用できます。

- Pod の名前
- Pod プロジェクト/namespace
- Pod のアノテーション
- Pod のラベル

手順

ボリュームプラグインを使用するには、以下の手順を実行します。

1. **volume-pod.yaml** ファイルを作成します。

```
kind: Pod
apiVersion: v1
metadata:
  labels:
    zone: us-east-coast
    cluster: downward-api-test-cluster1
    rack: rack-123
  name: dapi-volume-test-pod
  annotations:
    annotation1: "345"
    annotation2: "456"
spec:
  containers:
    - name: volume-test-container
      image: gcr.io/google_containers/busybox
      command: ["sh", "-c", "cat /tmp/etc/pod_labels /tmp/etc/pod_annotations"]
      volumeMounts:
        - name: podinfo
          mountPath: /tmp/etc
          readOnly: false
  volumes:
  - name: podinfo
    downwardAPI:
```

```
      defaultMode: 420
      items:
      - fieldRef:
          fieldPath: metadata.name
        path: pod_name
      - fieldRef:
          fieldPath: metadata.namespace
        path: pod_namespace
      - fieldRef:
          fieldPath: metadata.labels
        path: pod_labels
      - fieldRef:
          fieldPath: metadata.annotations
        path: pod_annotations
  restartPolicy: Never
```

2. **volume-pod.yaml** ファイルから Pod を作成します。

```
$ oc create -f volume-pod.yaml
```

3. コンテナーのログを確認し、設定されたフィールドの有無を確認します。

```
$ oc logs -p dapi-volume-test-pod
```

出力例

```
cluster=downward-api-test-cluster1
rack=rack-123
zone=us-east-coast
annotation1=345
annotation2=456
kubernetes.io/config.source=api
```

6.5.3. Downward API を使用してコンテナーリソースを使用する方法について

Pod の作成時に、Downward API を使用してコンピューティングリソースの要求および制限についての情報を挿入し、イメージおよびアプリケーションの作成者が特定の環境用のイメージを適切に作成できるようにします。

環境変数またはボリュームプラグインを使用してこれを実行できます。

6.5.3.1. 環境変数を使用したコンテナーリソースの使用

Pod を作成するときは、Downward API を使用し、環境変数を使ってコンピューティングリソースの要求と制限に関する情報を挿入できます。

手順

環境変数を使用するには、以下の手順を実行します。

1. Pod 設定の作成時に、**spec.container** フィールド内の **resources** フィールドの内容に対応する環境変数を指定します。

```
....
```

```
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox:1.24
      command: [ "/bin/sh", "-c", "env" ]
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      env:
        - name: MY_CPU_REQUEST
          valueFrom:
            resourceFieldRef:
              resource: requests.cpu
        - name: MY_CPU_LIMIT
          valueFrom:
            resourceFieldRef:
              resource: limits.cpu
        - name: MY_MEM_REQUEST
          valueFrom:
            resourceFieldRef:
              resource: requests.memory
        - name: MY_MEM_LIMIT
          valueFrom:
            resourceFieldRef:
              resource: limits.memory
....
```

リソース制限がコンテナー設定に含まれていない場合、Downward API はデフォルトでノードの CPU およびメモリーの割り当て可能な値に設定されます。

2. **pod.yaml** ファイルから Pod を作成します。

```
$ oc create -f pod.yaml
```

6.5.3.2. ボリュームプラグインを使用したコンテナーリソースの使用

Pod を作成するときは、Downward API を使用し、ボリュームプラグインを使ってコンピューティングリソースの要求と制限に関する情報を挿入できます。

手順

ボリュームプラグインを使用するには、以下の手順を実行します。

1. Pod 設定の作成時に、**spec.volumes.downwardAPI.items** フィールドを使用して **spec.resources** フィールドに対応する必要なリソースを記述します。

```
....
spec:
  containers:
    - name: client-container
      image: gcr.io/google_containers/busybox:1.24
      command: ["sh", "-c", "while true; do echo; if [[ -e /etc/cpu_limit ]]; then cat /etc/cpu_limit;
```

```
fi; if [[ -e /etc/cpu_request ]]; then cat /etc/cpu_request; fi; if [[ -e /etc/mem_limit ]]; then cat
/etc/mem_limit; fi; if [[ -e /etc/mem_request ]]; then cat /etc/mem_request; fi; sleep 5; done"]
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      volumeMounts:
        - name: podinfo
          mountPath: /etc
          readOnly: false
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "cpu_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.cpu
          - path: "cpu_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.cpu
          - path: "mem_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.memory
          - path: "mem_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.memory
....
```

リソース制限がコンテナー設定に含まれていない場合、Downward API はデフォルトでノードの CPU およびメモリーの割り当て可能な値に設定されます。

2. **volume-pod.yaml** ファイルから Pod を作成します。

```
$ oc create -f volume-pod.yaml
```

6.5.4. Downward API を使用したシークレットの使用

Pod の作成時に、Downward API を使用してシークレットを挿入し、イメージおよびアプリケーションの作成者が特定の環境用のイメージを作成できるようにできます。

手順

1. **secret.yaml** ファイルを作成します。

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
```

```
data:
  password: cGFzc3dvcmQ=
  username: ZGV2ZWxvcGVy
type: kubernetes.io/basic-auth
```

2. **secret.yaml** ファイルから **Secret** オブジェクトを作成します。

```
$ oc create -f secret.yaml
```

3. 上記の **Secret** オブジェクトから **username** フィールドを参照する **pod.yaml** ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: username
  restartPolicy: Never
```

4. **pod.yaml** ファイルから Pod を作成します。

```
$ oc create -f pod.yaml
```

5. コンテナーのログで **MY_SECRET_USERNAME** の値を確認します。

```
$ oc logs -p dapi-env-test-pod
```

6.5.5. Downward API を使用した設定マップの使用

Pod の作成時に、Downward API を使用して設定マップの値を挿入し、イメージおよびアプリケーションの作成者が特定の環境用のイメージを作成することができるようにすることができます。

手順

1. **configmap.yaml** ファイルを作成します。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfigmap
data:
  mykey: myvalue
```

2. **configmap.yaml** ファイルから **ConfigMap** オブジェクトを作成します。

```
$ oc create -f configmap.yaml
```

3. 上記の **ConfigMap** オブジェクトを参照する **pod.yaml** ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_CONFIGMAP_VALUE
          valueFrom:
            configMapKeyRef:
              name: myconfigmap
              key: mykey
  restartPolicy: Always
```

4. **pod.yaml** ファイルから Pod を作成します。

```
$ oc create -f pod.yaml
```

5. コンテナーのログで **MY_CONFIGMAP_VALUE** の値を確認します。

```
$ oc logs -p dapi-env-test-pod
```

6.5.6. 環境変数の参照

Pod の作成時に、**$()** 構文を使用して事前に定義された環境変数の値を参照できます。環境変数の参照が解決されない場合、値は提供された文字列のままになります。

手順

1. 既存の **environment variable** を参照する **pod.yaml** ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_EXISTING_ENV
          value: my_value
```

```
        - name: MY_ENV_VAR_REF_ENV
          value: $(MY_EXISTING_ENV)
  restartPolicy: Never
```

2. **pod.yaml** ファイルから Pod を作成します。

```
$ oc create -f pod.yaml
```

3. コンテナーのログで **MY_ENV_VAR_REF_ENV** 値を確認します。

```
$ oc logs -p dapi-env-test-pod
```

6.5.7. 環境変数の参照のエスケープ

Pod の作成時に、二重ドル記号を使用して環境変数の参照をエスケープできます。次に値は指定された値の単一ドル記号のバージョンに設定されます。

手順

1. 既存の **environment variable** を参照する **pod.yaml** ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_NEW_ENV
          value: $$(SOME_OTHER_ENV)
  restartPolicy: Never
```

2. **pod.yaml** ファイルから Pod を作成します。

```
$ oc create -f pod.yaml
```

3. コンテナーのログで **MY_NEW_ENV** 値を確認します。

```
$ oc logs -p dapi-env-test-pod
```

6.6. OPENSHIFT CONTAINER PLATFORM コンテナーへの/からのファイルのコピー

CLI を使用して、**rsync** コマンドでコンテナーのリモートディレクトリーにローカルファイルをコピーするか、またはそのディレクトリーからローカルファイルをコピーすることができます。

6.6.1. ファイルをコピーする方法について

oc rsync コマンドまたは remote sync は、バックアップと復元を実行するためにデータベースアーカ

イブを Pod にコピー、または Pod からコピーするのに役立つツールです。また、実行中の Pod がソースファイルのホットリロードをサポートする場合に、ソースコードの変更を開発のデバッグ目的で実行中の Pod にコピーするためにも、**oc rsync** を使用できます。

```
$ oc rsync <source> <destination> [-c <container>]
```

6.6.1.1. 要件

Copy Source の指定

oc rsync コマンドのソース引数はローカルディレクトリーまた Pod ディレクトリーのいずれかを示す必要があります。個々のファイルはサポートされていません。

Pod ディレクトリーを指定する場合、ディレクトリー名の前に Pod 名を付ける必要があります。

```
<pod name>:<dir>
```

ディレクトリー名がパスセパレーター (/) で終了する場合、ディレクトリーの内容のみが宛先にコピーされます。それ以外の場合は、ディレクトリーとその内容が宛先にコピーされます。

Copy Destination の指定

oc rsync コマンドの宛先引数はディレクトリーを参照する必要があります。ディレクトリーが存在せず、**rsync** がコピーに使用される場合、ディレクトリーが作成されます。

宛先でのファイルの削除

--delete フラグは、ローカルディレクトリーにないリモートディレクトリーにあるファイルを削除するために使用できます。

ファイル変更についての継続的な同期

--watch オプションを使用すると、コマンドはソースパスでファイルシステムの変更をモニターし、変更が生じるとそれらを同期します。この引数を指定すると、コマンドは無期限に実行されます。

同期は短い非表示期間の後に実行され、急速に変化するファイルシステムによって同期呼び出しが継続的に実行されないようにします。

--watch オプションを使用する場合、動作は通常 **oc rsync** に渡される引数の使用を含め **oc rsync** を繰り返し手動で起動する場合と同様になります。そのため、**--delete** などの **oc rsync** の手動の呼び出しで使用される同じフラグでこの動作を制御できます。

6.6.2. コンテナーへの/からのファイルのコピー

コンテナーへの/からのローカルファイルのコピーのサポートは CLI に組み込まれています。

前提条件

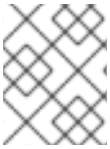
oc rsync を使用する場合は、以下の点に注意してください。

rsync がインストールされていること

oc rsync コマンドは、クライアントマシンおよびリモートコンテナー上に存在する場合は、ローカルの **rsync** ツールを使用します。

rsync がローカルの場所またはリモートコンテナーに見つからない場合は、**tar** アーカイブがローカルに作成されてからコンテナーに送信されます。ここで、**tar** ユーティリティーがファイルの展開に使用されます。リモートコンテナーで **tar** を利用できない場合は、コピーに失敗します。

tar のコピー方法は **oc rsync** と同様に機能する訳ではありません。たとえば、**oc rsync** は、宛先ディレクトリーが存在しない場合にはこれを作成し、ソースと宛先間の差分のファイルのみを送信します。



注記

Windows では、**cwRsync** クライアントが **oc rsync** コマンドで使用するためにインストールされ、PATH に追加される必要があります。

手順

- ローカルディレクトリーを Pod ディレクトリーにコピーするには、以下の手順を実行します。

```
$ oc rsync <local-dir> <pod-name>:/<remote-dir> -c <container-name>
```

以下に例を示します。

```
$ oc rsync /home/user/source devpod1234:/src -c user-container
```

- Pod ディレクトリーをローカルディレクトリーにコピーするには、以下の手順を実行します。

```
$ oc rsync devpod1234:/src /home/user/source
```

出力例

```
$ oc rsync devpod1234:/src/status.txt /home/user/
```

6.6.3. 高度な Rsync 機能の使用

oc rsync コマンドは標準の **rsync** よりも少ないコマンドラインのオプションを表示します。**oc rsync** で利用できない標準の **rsync** コマンドラインオプションを使用する必要がある場合 (例: **--exclude-from=FILE** オプション)、以下のように回避策として標準 **rsync** の **--rsh** (**-e**) オプション、または **RSYNC_RSH** 環境変数を使用できる場合があります。

```
$ rsync --rsh='oc rsh' --exclude-from=FILE SRC POD:DEST
```

または、以下を実行します。

RSYNC_RSH 変数をエクスポートします。

```
$ export RSYNC_RSH='oc rsh'
```

次に、rsync コマンドを実行します。

```
$ rsync --exclude-from=FILE SRC POD:DEST
```

上記の例のいずれも標準の **rsync** をリモートシェルプログラムとして **oc rsh** を使用するように設定して リモート Pod に接続できるようにします。これらは **oc rsync** を実行する代替方法となります。

6.7. OPENSHIFT CONTAINER PLATFORM コンテナーでのリモートコマンドの実行

OpenShift Container Platform コンテナーでリモートコマンドを実行するために、CLI を使用することができます。

6.7.1. コンテナーでのリモートコマンドの実行

リモートコンテナーコマンドの実行についてサポートは CLI に組み込まれています。

手順

コンテナーでコマンドを実行するには、以下の手順を実行します。

```
$ oc exec <pod> [-c <container>] <command> [<arg_1> ... <arg_n>]
```

以下に例を示します。

```
$ oc exec mypod date
```

出力例

```
Thu Apr  9 02:21:53 UTC 2015
```

重要

セキュリティー保護の理由 により、**oc exec** コマンドは、コマンドが **cluster-admin** ユーザーによって実行されている場合を除き、特権付きコンテナーにアクセスしようとしても機能しません。

6.7.2. クライアントからのリモートコマンドを開始するためのプロトコル

クライアントは要求を Kubernetes API サーバーに対して実行してコンテナーのリモートコマンドの実行を開始します。

```
/proxy/nodes/<node_name>/exec/<namespace>/<pod>/<container>?command=<command>
```

上記の URL には以下が含まれます。

- **<node_name>** はノードの FQDN です。
- **<namespace>** はターゲット Pod のプロジェクトです。
- **<pod>** はターゲット Pod の名前です。
- **<container>** はターゲットコンテナーの名前です。
- **<command>** は実行される必要なコマンドです。

以下に例を示します。

```
/proxy/nodes/node123.openshift.com/exec/myns/mypod/mycontainer?command=date
```

さらに、クライアントはパラメーターを要求に追加して以下について指示します。

- クライアントはリモートクライアントのコマンドに入力を送信する (標準入力: stdin)。

- クライアントのターミナルは TTY である。
- リモートコンテナーのコマンドは標準出力 (stdout) からクライアントに出力を送信する。
- リモートコンテナーのコマンドは標準エラー出力 (stderr) からクライアントに出力を送信する。

exec 要求の API サーバーへの送信後、クライアントは多重化ストリームをサポートするものに接続をアップグレードします。現在の実装では **HTTP/2** を使用しています。

クライアントは標準入力 (stdin)、標準出力 (stdout)、および標準エラー出力 (stderr) 用にそれぞれのストリームを作成します。ストリームを区別するために、クライアントはストリームの **streamType** ヘッダーを **stdin**、**stdout**、または **stderr** のいずれかに設定します。

リモートコマンド実行要求の処理が終了すると、クライアントはすべてのストリームやアップグレードされた接続および基礎となる接続を閉じます。

6.8. コンテナー内のアプリケーションにアクセスするためのポート転送の使用

OpenShift Container Platform は、Pod へのポート転送をサポートします。

6.8.1. ポート転送について

CLI を使用して 1 つ以上のローカルポートを Pod に転送できます。これにより、指定されたポートまたはランダムのポートでローカルにリッスンでき、Pod の所定ポートへ/からデータを転送できます。

ポート転送のサポートは、CLI に組み込まれています。

```
$ oc port-forward <pod> [<local_port>:]<remote_port> [...[<local_port_n>:]<remote_port_n>]
```

CLI はユーザーによって指定されたそれぞれのローカルポートでリッスンし、以下で説明されているプロトコルで転送を実行します。

ポートは以下の形式を使用して指定できます。

5000	クライアントはポート 5000 でローカルにリッスンし、Pod の 5000 に転送します。
6000:5000	クライアントはポート 6000 でローカルにリッスンし、Pod の 5000 に転送します。
:5000 または **0:5000**	クライアントは空きのローカルポートを選択し、Pod の 5000 に転送します。

OpenShift Container Platform は、クライアントからのポート転送要求を処理します。要求を受信すると、OpenShift Container Platform は応答をアップグレードし、クライアントがポート転送ストリームを作成するまで待機します。OpenShift Container Platform が新規ストリームを受信したら、ストリームと Pod のポート間でデータをコピーします。

アーキテクチャーの観点では、Pod のポートに転送するためのいくつかのオプションがあります。サポートされている OpenShift Container Platform 実装はノードホストで直接 **nsenter** を直接呼び出して、Pod ネットワークの namespace に入ってから、**socat** を呼び出してストリームと Pod のポート間

でデータをコピーします。ただし、カスタムの実装には、**nsenter** および **socat** を実行する **helper** Pod の実行を含めることができ、その場合は、それらのバイナリーをホストにインストールする必要はありません。

6.8.2. ポート転送の使用

CLI を使用して、1 つ以上のローカルポートの Pod へのポート転送を実行できます。

手順

以下のコマンドを使用して、Pod 内の指定されたポートでリッスンします。

```
$ oc port-forward <pod> [<local_port>:]<remote_port> [...[<local_port_n>:]<remote_port_n>]
```

以下に例を示します。

- 以下のコマンドを使用して、ポート **5000** および **6000** でローカルにリッスンし、Pod のポート **5000** および **6000** との間でデータを転送します。

  ```
  $ oc port-forward <pod> 5000 6000
  ```

  出力例

  ```
  Forwarding from 127.0.0.1:5000 -> 5000
  Forwarding from [::1]:5000 -> 5000
  Forwarding from 127.0.0.1:6000 -> 6000
  Forwarding from [::1]:6000 -> 6000
  ```

- 以下のコマンドを使用して、ポート **8888** でローカルにリッスンし、Pod の **5000** に転送します。

  ```
  $ oc port-forward <pod> 8888:5000
  ```

  出力例

  ```
  Forwarding from 127.0.0.1:8888 -> 5000
  Forwarding from [::1]:8888 -> 5000
  ```

- 以下のコマンドを使用して、空きポートでローカルにリッスンし、Pod の **5000** に転送します。

  ```
  $ oc port-forward <pod> :5000
  ```

  出力例

  ```
  Forwarding from 127.0.0.1:42390 -> 5000
  Forwarding from [::1]:42390 -> 5000
  ```

  または、以下を実行します。

  ```
  $ oc port-forward <pod> 0:5000
  ```

6.8.3. クライアントからのポート転送を開始するためのプロトコル

クライアントは Kubernetes API サーバーに対して要求を実行して Pod へのポート転送を実行します。

```
/proxy/nodes/<node_name>/portForward/<namespace>/<pod>
```

上記の URL には以下が含まれます。

- **<node_name>** はノードの FQDN です。
- **<namespace>** はターゲット Pod の namespace です。
- **<pod>** はターゲット Pod の名前です。

以下に例を示します。

```
/proxy/nodes/node123.openshift.com/portForward/myns/mypod
```

ポート転送要求を API サーバーに送信した後に、クライアントは多重化ストリームをサポートするものに接続をアップグレードします。現在の実装では Hyptertext Transfer Protocol Version 2 (HTTP/2) を使用しています。

クライアントは Pod のターゲットポートを含む **port** ヘッダーでストリームを作成します。ストリームに書き込まれるすべてのデータは kubelet 経由でターゲット Pod およびポートに送信されます。同様に、転送された接続で Pod から送信されるすべてのデータはクライアントの同じストリームに送信されます。

クライアントは、ポート転送要求が終了するとすべてのストリーム、アップグレードされた接続および基礎となる接続を閉じます。

6.9. コンテナーでの SYSCTL の使用

sysctl 設定は Kubernetes 経由で公開され、ユーザーがコンテナー内の namespace の特定のカーネルパラメーターをランタイム時に変更できるようにします。namespace を使用する sysctl のみを Pod 上で独立して設定できます。sysctl に namespace がない場合 (**ノードレベル**と呼ばれる)、Node Tuning Operator など、sysctl を設定する別の方法を使用する必要があります。さらに **安全** とみなされる sysctl のみがデフォルトでホワイトリストに入れられます。 他の **安全でない** sysctl はノードで手動で有効にし、ユーザーが使用できるようにできます。

6.9.1. sysctl について

Linux では、管理者は sysctl インターフェイスを使ってランタイム時にカーネルパラメーターを変更することができます。パラメーターは **/proc/sys/** 仮想プロセスファイルシステムで利用できます。これらのパラメーターは以下を含む各種のサブシステムを対象とします。

- カーネル (共通の接頭辞: **kernel.**)
- ネットワーク (共通の接頭辞: **net.**)
- 仮想メモリー (共通の接頭辞: **vm.**)
- MDADM (共通の接頭辞: **dev.**)

追加のサブシステムについては、カーネルのドキュメント で説明されています。すべてのパラメーターの一覧を表示するには、以下のコマンドを実行します。

```
$ sudo sysctl -a
```

6.9.1.1. namespace を使用した sysctl vs ノードレベルの sysctl

Linux カーネルでは、数多くの sysctl に **namespace** が使用されています。これは、それらをノードの各 Pod に対して個別に設定できることを意味します。namespace の使用は、sysctl を Kubernetes 内の Pod 環境でアクセス可能にするための要件になります。

以下の sysctl は namespace を使用するものとして知られている sysctl です。

- **kernel.shm***
- **kernel.msg***
- **kernel.sem**
- **fs.mqueue.***

また、**net.*** グループの大半の sysctl には namespace が使用されていることが知られています。それらの namespace の使用は、カーネルのバージョンおよびディストリビューターによって異なります。

namespace が使用されていない sysctl は **ノードレベル** と呼ばれており、クラスター管理者がノードの基礎となる Linux ディストリビューションを使用 (例: **/etc/sysctls.conf** ファイルを変更) するか、または特権付きコンテナーでデーモンセットを使用することによって手動で設定する必要があります。Node Tuning Operator を使用して **node-level** を設定できます。

注記

特殊な sysctl が設定されたノードにテイントのマークを付けることを検討してください。それらの sysctl 設定を必要とするノードにのみ Pod をスケジュールします。テイントおよび容認 (Toleration) 機能を使用してノードにマークを付けます。

6.9.1.2. 安全 vs 安全でない sysctl

sysctl は **安全な** および **安全でない** sysctl に分類されます。

sysctl が安全であるとみなされるには、適切な namespace を使用し、同じノード上の Pod 間で適切に分離する必要があります。Pod ごとに sysctl を設定する場合は、以下の点に留意してください。

- この設定はノードのその他の Pod に影響を与えないものである。
- この設定はノードの正常性に負の影響を与えないものである。
- この設定は Pod のリソース制限を超える CPU またはメモリーリソースの取得を許可しないものである。

OpenShift Container Platform は以下の sysctl を安全なセットでサポートするか、またはホワイトリスト化します。

- **kernel.shm_rmid_forced**
- **net.ipv4.ip_local_port_range**
- **net.ipv4.tcp_syncookies**
- **net.ipv4.ping_group_range**

すべての安全な sysctl はデフォルトで有効にされます。**Pod** 仕様を変更して、Pod で sysctl を使用できます。

OpenShift Container Platform でホワイトリスト化されない sysctl は OpenShift Container Platform で安全でないと見なされます。namespace を使用するだけで、sysctl が安全であるとみなされる訳ではありません。

すべての安全でない sysctl はデフォルトで無効にされ、ノードごとにクラスター管理者によって手動で有効にされる必要があります。無効にされた安全でない sysctl が設定された Pod はスケジュールされますが、起動されません。

```
$ oc get pod
```

出力例

```
NAME       READY   STATUS          RESTARTS   AGE
hello-pod   0/1     SysctlForbidden   0          14s
```

6.9.2. Pod の sysctl 設定

Pod の **securityContext** を使用して sysctl を Pod に設定できます。 **securityContext** は同じ Pod 内のすべてのコンテナーに適用されます。

安全な sysctl はデフォルトで許可されます。安全でない sysctl が設定された Pod は、クラスター管理者がそのノードの安全でない sysctl を明示的に有効にしない限り、いずれのノードでも起動に失敗します。ノードレベルの sysctl の場合のように、それらの Pod を正しいノードにスケジュールするには、テイントおよび容認 (Toleration)、またはノードのラベルを使用します。

以下の例では Pod の **securityContext** を使用して安全な sysctl **kernel.shm_rmid_forced** および 2 つの安全でない sysctl **net.core.somaxconn** および **kernel.msgmax** を設定します。仕様では **安全な** sysctl と **安全でない** sysctl は区別されません。

警告

オペレーティングシステムが不安定になるのを防ぐには、変更の影響を確認している場合にのみ sysctl パラメーターを変更します。

手順

安全なおよび安全でない sysctl を使用するには、以下を実行します。

1. 以下の例に示されるように、Pod を定義する YAML ファイルを変更し、**securityContext** 仕様を追加します。

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example
spec:
  securityContext:
```

```
    sysctls:
    - name: kernel.shm_rmid_forced
      value: "0"
    - name: net.core.somaxconn
      value: "1024"
    - name: kernel.msgmax
      value: "65536"
  ...
```

2. Pod を作成します。

```
$ oc apply -f <file-name>.yaml
```

安全でない sysctl がノードに許可されていない場合、Pod はスケジュールされますが、デプロイはされません。

```
$ oc get pod
```

出力例

```
NAME       READY   STATUS            RESTARTS   AGE
hello-pod   0/1     SysctlForbidden   0          14s
```

6.9.3. 安全でない sysctl の有効化

クラスター管理者は、高パフォーマンスまたはリアルタイムのアプリケーション調整などの非常に特殊な状況で特定の安全でない sysctl を許可することができます。

安全でない sysctl を使用する必要がある場合、クラスター管理者は特定のタイプのノードに対してそれらを個別に有効にする必要があります。sysctl には namespace を使用する必要があります。

SCC (Security Context Constraints) の **forbiddenSysctls** および **allowedUnsafeSysctls** フィールドに sysctl または sysctl パターンの一覧を指定して、Pod に設定できる sysctl をさらに制御できます。

- **forbiddenSysctls** オプションは、特定の sysctl を除外します。
- **allowedUnsafeSysctls** オプションは、高パフォーマンスやリアルタイムのアプリケーションチューニングなどの特定ニーズを管理します。

警告

安全でないという性質上、安全でない sysctl は各自の責任で使用されます。 場合によっては、コンテナーの正しくない動作やリソース不足、またはノードの破損などの深刻な問題が生じる可能性があります。

手順

1. ラベルを安全でない sysctl が設定されたコンテナーが実行されるマシン設定プールに追加します。

```
$ oc edit machineconfigpool worker
```

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  creationTimestamp: 2019-02-08T14:52:39Z
  generation: 1
  labels:
    custom-kubelet: sysctl (1)
```

(1) **key: pair** ラベルを追加します。

2. **KubeletConfig** カスタムリソース (CR) を作成します。

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: custom-kubelet
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: sysctl (1)
  kubeletConfig:
    allowedUnsafeSysctls: (2)
      - "kernel.msg*"
      - "net.core.somaxconn"
```

(1) マシン設定プールからラベルを指定します。

(2) 許可する必要のある安全でない sysctl を一覧表示します。

3. オブジェクトを作成します。

```
$ oc apply -f set-sysctl-worker.yaml
```

99-worker-XXXXXX-XXXXX-XXXX-XXXXX-kubelet 形式で指定される新規の **MachineConfig** オブジェクトが作成されます。

4. **machineconfigpool** オブジェクト **ステータス** フィールドを使用してクラスターが再起動するまで待機します。
以下に例を示します。

```
status:
  conditions:
    - lastTransitionTime: '2019-08-11T15:32:00Z'
      message: >-
        All nodes are updating to
        rendered-worker-ccbfb5d2838d65013ab36300b7b3dc13
      reason: ''
      status: 'True'
      type: Updating
```

クラスターの準備ができると、以下のようなメッセージが表示されます。

```
- lastTransitionTime: '2019-08-11T16:00:00Z'
  message: >-
    All nodes are updated with
    rendered-worker-ccbfb5d2838d65013ab36300b7b3dc13
  reason: ''
  status: 'True'
  type: Updated
```

5. クラスターが準備状態になる場合、新規 **MachineConfig** オブジェクトでマージされた **KubeletConfig** オブジェクトを確認します。

```
$ oc get machineconfig 99-worker-XXXXXX-XXXXX-XXXX-XXXXX-kubelet -o json | grep ownerReference -A7
```

```
    "ownerReferences": [
        {
            "apiVersion": "machineconfiguration.openshift.io/v1",
            "blockOwnerDeletion": true,
            "controller": true,
            "kind": "KubeletConfig",
            "name": "custom-kubelet",
            "uid": "3f64a766-bae8-11e9-abe8-0a1a2a4813f2"
        }
    ]
```

安全でない sysctl を必要に応じて Pod に追加することができるようになります。

第7章 クラスターの操作

7.1. OPENSHIFT CONTAINER PLATFORM クラスター内のシステムイベント情報の表示

OpenShift Container Platform のイベントは OpenShift Container Platform クラスターの API オブジェクトに対して発生するイベントに基づいてモデル化されます。

7.1.1. イベントについて

イベントにより、OpenShift Container Platform はリソースに依存しない方法で実際のイベントについての情報を記録できます。また、開発者および管理者が統一された方法でシステムコンポーネントについての情報を使用できるようにします。

7.1.2. CLI を使用したイベントの表示

CLI を使用し、特定のプロジェクト内のイベントの一覧を取得できます。

手順

- プロジェクト内のイベントを表示するには、以下のコマンドを使用します。

```
$ oc get events [-n <project>] 1
```

1 プロジェクトの名前。

以下に例を示します。

```
$ oc get events -n openshift-config
```

出力例

```
LAST SEEN   TYPE      REASON                  OBJECT                     MESSAGE
97m         Normal    Scheduled               pod/dapi-env-test-pod      Successfully assigned
openshift-config/dapi-env-test-pod to ip-10-0-171-202.ec2.internal
97m         Normal    Pulling                 pod/dapi-env-test-pod      pulling image
"gcr.io/google_containers/busybox"
97m         Normal    Pulled                  pod/dapi-env-test-pod      Successfully pulled image
"gcr.io/google_containers/busybox"
97m         Normal    Created                 pod/dapi-env-test-pod      Created container
9m5s        Warning   FailedCreatePodSandBox  pod/dapi-volume-test-pod   Failed create
pod sandbox: rpc error: code = Unknown desc = failed to create pod network sandbox
k8s_dapi-volume-test-pod_openshift-config_6bc60c1f-452e-11e9-9140-
0eec59c23068_0(748c7a40db3d08c07fb4f9eba774bd5effe5f0d5090a242432a73eee66ba9e22
): Multus: Err adding pod to network "openshift-sdn": cannot set "openshift-sdn" ifname to
"eth0": no netns: failed to Statfs "/proc/33366/ns/net": no such file or directory
8m31s       Normal    Scheduled               pod/dapi-volume-test-pod   Successfully assigned
openshift-config/dapi-volume-test-pod to ip-10-0-171-202.ec2.internal
```

- OpenShift Container Platform コンソールからプロジェクト内のイベントを表示するには、以下を実行します。

1. OpenShift Container Platform コンソールを起動します。
2. **Home → Events** をクリックし、プロジェクトを選択します。
3. イベントを表示するリソースに移動します。たとえば、**Home → Projects →** <project-name> → <resource-name> の順に移動します。
   Pod や デプロイメントなどの多くのオブジェクトには、独自の **イベント** タブもあります。 それらのタブには、オブジェクトに関連するイベントが表示されます。

7.1.3. イベントの一覧

このセクションでは、OpenShift Container Platform のイベントについて説明します。

表7.1 設定イベント

名前	説明
FailedValidation	Pod 設定の検証に失敗しました。

表7.2 コンテナーイベント

名前	説明
BackOff	バックオフ (再起動) によりコンテナーが失敗しました。
Created	コンテナーが作成されました。
Failed	プル/作成/起動が失敗しました。
Killing	コンテナーを強制終了しています。
Started	コンテナーが起動しました。
Preempting	他の Pod のプリエンプションを実行します。
ExceededGracePeriod	コンテナーランタイムは、指定の猶予期間以内に Pod を停止しませんでした。

表7.3 正常性に関するイベント

名前	説明
Unhealthy	コンテナーが正常ではありません。

表7.4 イメージイベント

名前	説明
BackOff	バックオフ (コンテナー起動、イメージのプル)。
ErrImageNeverPull	イメージの NeverPull Policy の違反があります。
Failed	イメージのプルに失敗しました。
InspectFailed	イメージの検査に失敗しました。
Pulled	イメージのプルに成功し、コンテナーイメージがマシンにすでに置かれています。
Pulling	イメージをプルしています。

表7.5 イメージマネージャーイベント

名前	説明
FreeDiskSpaceFailed	空きディスク容量に関連する障害が発生しました。
InvalidDiskCapacity	無効なディスク容量です。

表7.6 ノードイベント

名前	説明
FailedMount	ボリュームのマウントに失敗しました。
HostNetworkNotSupported	ホストのネットワークがサポートされていません。
HostPortConflict	ホスト/ポートの競合
KubeletSetupFailed	Kubelet のセットアップに失敗しました。
NilShaper	シェイパーが定義されていません。
NodeNotReady	ノードの準備ができていません。
NodeNotSchedulable	ノードがスケジュール可能ではありません。

名前	説明
NodeReady	ノードの準備ができています。
NodeSchedulable	ノードがスケジュール可能です。
NodeSelectorMismatching	ノードセレクターの不一致があります。
OutOfDisk	ディスクの空き容量が不足しています。
Rebooted	ノードが再起動しました。
Starting	kubelet を起動しています。
FailedAttachVolume	ボリュームの割り当てに失敗しました。
FailedDetachVolume	ボリュームの割り当て解除に失敗しました。
VolumeResizeFailed	ボリュームの拡張/縮小に失敗しました。
VolumeResizeSuccessful	正常にボリュームを拡張/縮小しました。
FileSystemResizeFailed	ファイルシステムの拡張/縮小に失敗しました。
FileSystemResizeSuccessful	正常にファイルシステムが拡張/縮小されました。
FailedUnMount	ボリュームのマウント解除に失敗しました。
FailedMapVolume	ボリュームのマッピングに失敗しました。
FailedUnmapDevice	デバイスのマッピング解除に失敗しました。
AlreadyMountedVolume	ボリュームがすでにマウントされています。
SuccessfulDetachVolume	ボリュームの割り当てが正常に解除されました。

名前	説明
SuccessfulMountVolume	ボリュームが正常にマウントされました。
SuccessfulUnMountVolume	ボリュームのマウントが正常に解除されました。
ContainerGCFailed	コンテナーのガベージコレクションに失敗しました。
ImageGCFailed	イメージのガベージコレクションに失敗しました。
FailedNodeAllocatableEnforcement	システム予約の Cgroup 制限の実施に失敗しました。
NodeAllocatableEnforced	システム予約の Cgroup 制限を有効にしました。
UnsupportedMountOption	マウントオプションが非対応です。
SandboxChanged	Pod のサンドボックスが変更されました。
FailedCreatePodSandBox	Pod のサンドボックスの作成に失敗しました。
FailedPodSandBoxStatus	Pod サンドボックスの状態取得に失敗しました。

表7.7 Pod ワーカーイベント

名前	説明
FailedSync	Pod の同期が失敗しました。

表7.8 システムイベント

名前	説明
SystemOOM	クラスターに OOM (out of memory) 状態が発生しました。

表7.9 Pod に関するイベント

名前	説明
FailedKillPod	Pod の停止に失敗しました。
FailedCreatePodContainer	Pod コンテナーの作成に失敗しました。
Failed	Pod データディレクトリーの作成に失敗しました。
NetworkNotReady	ネットワークの準備ができていません。
FailedCreate	作成エラー: **<error-msg>**
SuccessfulCreate	作成された Pod: **<pod-name>**
FailedDelete	削除エラー: **<error-msg>**
SuccessfulDelete	削除した Pod: **<pod-id>**

表7.10 Horizontal Pod AutoScaler に関するイベント

名前	説明
SelectorRequired	セレクターが必要です。
InvalidSelector	セレクターを適切な内部セレクターオブジェクトに変換できませんでした。
FailedGetObjectMetric	HPA はレプリカ数を計算できませんでした。
InvalidMetricSourceType	不明なメトリクスソースタイプです。
ValidMetricFound	HPA は正常にレプリカ数を計算できました。
FailedConvertHPA	指定の HPA への変換に失敗しました。
FailedGetScale	HPA コントローラーは、ターゲットの現在のスケーリングを取得できませんでした。
SucceededGetScale	HPA コントローラーは、ターゲットの現在のスケールを取得できました。

名前	説明
FailedComputeMetricsReplicas	表示されているメトリクスに基づく必要なレプリカ数の計算に失敗しました。
FailedRescale	新しいサイズ: **<size>**; 理由:**<msg>**; エラー:**<error-msg>**
SuccessfulRescale	新しいサイズ: **<size>**; 理由:**<msg>**.
FailedUpdateStatus	状況の更新に失敗しました。

表7.11 ネットワークイベント (openshift-sdn)

名前	説明
Starting	OpenShift-SDN を開始します。
NetworkFailed	Pod のネットワークインターフェイスがなくなり、Pod が停止します。

表7.12 ネットワークイベント (kube-proxy)

名前	説明
NeedPods	サービスポート **<serviceName>:<port>** は Pod が必要です。

表7.13 ボリュームイベント

名前	説明
FailedBinding	利用可能な永続ボリュームがなく、ストレージクラスが設定されていません。
VolumeMismatch	ボリュームサイズまたはクラスが要求の内容と異なります。
VolumeFailedRecycle	再利用 Pod の作成エラー
VolumeRecycled	ボリュームの再利用時に発生します。
RecyclerPod	Pod の再利用時に発生します。
VolumeDelete	ボリュームの削除時に発生します。

名前	説明
VolumeFailedDelete	ボリュームの削除時のエラー。
ExternalProvisioning	要求のボリュームが手動または外部ソフトウェアでプロビジョニングされる場合に発生します。
ProvisioningFailed	ボリュームのプロビジョニングに失敗しました。
ProvisioningCleanupFailed	プロビジョニングしたボリュームの消去エラー
ProvisioningSucceeded	ボリュームが正常にプロビジョニングされる場合に発生します。
WaitForFirstConsumer	Pod のスケジューリングまでバインドが遅延します。

表7.14 ライフサイクルフック

名前	説明
FailedPostStartHook	ハンドラーが Pod の起動に失敗しました。
FailedPreStopHook	ハンドラーが pre-stop に失敗しました。
UnfinishedPreStopHook	Pre-stop フックが完了しませんでした。

表7.15 デプロイメント

名前	説明
DeploymentCancellationFailed	デプロイメントのキャンセルに失敗しました。
DeploymentCancelled	デプロイメントがキャンセルされました。
DeploymentCreated	新規レプリケーションコントローラーが作成されました。

名前	説明
IngressIPRangeFull	サービスに割り当てる Ingress IP がありません。

表7.16 スケジューラーイベント

名前	説明
FailedScheduling	Pod のスケジューリングに失敗: **<pod-namespace>/<pod-name>**。このイベントは **AssumePodVolumes** の失敗、バインドの拒否など、複数の理由で発生します。
Preempted	ノード **<node-name>** にある **<preemptor-namespace>/<preemptor-name>**
Scheduled	**<node-name>**. に **<pod-name>** が正常に割り当てられました。

表7.17 デーモンセットイベント

名前	説明
SelectingAll	この DaemonSet は全 Pod を選択しています。空でないセレクターが必要です。
FailedPlacement	**<node-name>** への Pod の配置に失敗しました。
FailedDaemonPod	ノード **<node-name>** で問題のあるデーモン Pod**<pod-name>** が見つかりました。この Pod の終了を試行します。

表7.18 LoadBalancer サービスイベント

名前	説明
CreatingLoadBalancerFailed	ロードバランサーの作成エラー
DeletingLoadBalancer	ロードバランサーを削除します。
EnsuringLoadBalancer	ロードバランサーを確保します。
EnsuredLoadBalancer	ロードバランサーを確保しました。
UnAvailableLoadBalancer	**LoadBalancer** サービスに利用可能なノードがありません。

名前	説明
LoadBalancerSourceRanges	新規の **LoadBalancerSourceRanges** を表示します。例: **<old-source-range> → <new-source-range>**
LoadbalancerIP	新しい IP アドレスを表示します。例: **<old-ip> → <new-ip>**
ExternalIP	外部 IP アドレスを表示します。例: **Added: <external-ip>**
UID	新しい UID を表示します。例: **<old-service-uid> → <new-service-uid>**
ExternalTrafficPolicy	新しい **ExternalTrafficPolicy** を表示します。例: **<old-policy> → <new-policy>**
HealthCheckNodePort	新しい **HealthCheckNodePort** を表示します。例: **<old-node-port> → new-node-port>**
UpdatedLoadBalancer	新規ホストでロードバランサーを更新しました。
LoadBalancerUpdateFailed	新規ホストでのロードバランサーの更新に失敗しました。
DeletingLoadBalancer	ロードバランサーを削除します。
DeletingLoadBalancerFailed	ロードバランサーの削除エラー。
DeletedLoadBalancer	ロードバランサーを削除しました。

7.2. OPENSHIFT CONTAINER PLATFORM のノードが保持できる POD の数の見積り

クラスター管理者は、クラスター容量ツールを使用して、現在のリソースが使い切られる前にそれらを増やすべくスケジュール可能な Pod 数を表示し、スケジュール可能な Pod 数を表示したり、Pod を今後スケジュールできるようにすることができます。この容量は、クラスター内の個別ノードからのものを集めたものであり、これには CPU、メモリー、ディスク領域などが含まれます。

7.2.1. OpenShift Container Platform クラスター容量ツールについて

クラスター容量ツールはより正確な見積もりを出すべく、スケジュールの一連の意思決定をシミュレーションし、リソースが使い切られる前にクラスターでスケジュールできる入力 Pod のインスタンス数を判別します。

注記

ノード間に分散しているすべてのリソースがカウントされないため、残りの割り当て可能な容量は概算となります。残りのリソースのみが分析対象となり、クラスターでのスケジュール可能な所定要件を持つ Pod のインスタンス数という点から消費可能な容量を見積もります。

Pod のスケジューリングはその選択およびアフィニティー条件に基づいて特定のノードセットについてのみサポートされる可能性があります。そのため、クラスターでスケジュール可能な残りの Pod 数を見積もることが困難になる場合があります。

クラスター容量分析ツールは、コマンドラインからスタンドアロンのユーティリティーとして実行することも、OpenShift Container Platform クラスター内の Pod でジョブとして実行することもできます。これを Pod 内のジョブとして実行すると、介入なしに複数回実行することができます。

7.2.2. コマンドラインでのクラスター容量ツールの実行

コマンドラインから OpenShift Container Platform クラスター容量ツールを実行して、クラスターにスケジュール設定可能な Pod 数を見積ることができます。

前提条件

- OpenShift Cluster Capacity Tool を実行します。これは、RedHat エコシステムカタログからコンテナーイメージとして入手できます。
- ツールがリソース使用状況を見積もるために使用するサンプル **Pod** 仕様ファイルを作成します。**podspec** はそのリソース要件を **limits** または **requests** として指定します。クラスター容量ツールは、Pod のリソース要件をその見積もりの分析に反映します。
  Pod 仕様入力の例は以下の通りです。

```
apiVersion: v1
kind: Pod
metadata:
  name: small-pod
  labels:
    app: guestbook
    tier: frontend
spec:
  containers:
  - name: php-redis
    image: gcr.io/google-samples/gb-frontend:v4
    imagePullPolicy: Always
    resources:
      limits:
        cpu: 150m
        memory: 100Mi
      requests:
        cpu: 150m
        memory: 100Mi
```

手順

コマンドラインでクラスター容量ツールを使用するには、次のようにします。

1. ターミナルから、RedHat レジストリーにログインします。

```
$ podman login registry.redhat.io
```

2. クラスター容量ツールのイメージをプルします。

```
$ podman pull registry.redhat.io/openshift4/ose-cluster-capacity
```

3. クラスター容量ツールを実行します。

```
$ podman run -v $HOME/.kube:/kube:Z -v $(pwd):/cc:Z  ose-cluster-capacity \
/bin/cluster-capacity --kubeconfig /kube/config --podspec /cc/pod-spec.yaml \
--verbose (1)
```

(1) **--verbose** オプションを追加して、クラスター内の各ノードでスケジュールできる Pod の数の詳細な説明を出力することもできます。

出力例

```
small-pod pod requirements:
 - CPU: 150m
 - Memory: 100Mi

The cluster can schedule 88 instance(s) of the pod small-pod.

Termination reason: Unschedulable: 0/5 nodes are available: 2 Insufficient cpu,
3 node(s) had taint {node-role.kubernetes.io/master: }, that the pod didn't
tolerate.

Pod distribution among nodes:
small-pod
 - 192.168.124.214: 45 instance(s)
 - 192.168.124.120: 43 instance(s)
```

上記の例では、クラスターにスケジュールできる推定 Pod の数は 88 です。

7.2.3. Pod 内のジョブとしてのクラスター容量ツールの実行

クラスター容量ツールを Pod 内のジョブとして実行すると、ユーザーの介入なしに複数回実行できるという利点があります。クラスター容量ツールをジョブとして実行するには、**ConfigMap** オブジェクトを使用する必要があります。

前提条件

cluster-capacity ツール をダウンロードし、これをインストールします。

手順

クラスター容量ツールを実行するには、以下の手順を実行します。

1. クラスターロールを作成します。

```
$ cat << EOF| oc create -f -
```

出力例

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: cluster-capacity-role
rules:
- apiGroups: [""]
  resources: ["pods", "nodes", "persistentvolumeclaims", "persistentvolumes", "services",
"replicationcontrollers"]
  verbs: ["get", "watch", "list"]
- apiGroups: ["apps"]
  resources: ["replicasets", "statefulsets"]
  verbs: ["get", "watch", "list"]
- apiGroups: ["policy"]
  resources: ["poddisruptionbudgets"]
  verbs: ["get", "watch", "list"]
- apiGroups: ["storage.k8s.io"]
  resources: ["storageclasses"]
  verbs: ["get", "watch", "list"]
EOF
```

2. サービスアカウントを作成します。

```
$ oc create sa cluster-capacity-sa
```

3. ロールをサービスアカウントに追加します。

```
$ oc adm policy add-cluster-role-to-user cluster-capacity-role \
    system:serviceaccount:default:cluster-capacity-sa
```

4. **Pod** 仕様を定義し、作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: small-pod
  labels:
    app: guestbook
    tier: frontend
spec:
  containers:
  - name: php-redis
    image: gcr.io/google-samples/gb-frontend:v4
    imagePullPolicy: Always
    resources:
      limits:
        cpu: 150m
        memory: 100Mi
      requests:
        cpu: 150m
        memory: 100Mi
```

5. クラスター容量分析は、**cluster-capacity-configmap** という名前の **ConfigMap** オブジェクトを使用してボリュームにマウントされ、入力 Pod 仕様ファイル **pod.yaml** はパス **/test-pod** のボリューム **test-volume** にマウントされます。

ConfigMap オブジェクトを作成していない場合は、ジョブの作成前にこれを作成します。

```
$ oc create configmap cluster-capacity-configmap \
    --from-file=pod.yaml=pod.yaml
```

6. ジョブ仕様ファイルの以下のサンプルを使用して、ジョブを作成します。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: cluster-capacity-job
spec:
  parallelism: 1
  completions: 1
  template:
    metadata:
      name: cluster-capacity-pod
    spec:
        containers:
        - name: cluster-capacity
          image: openshift/origin-cluster-capacity
          imagePullPolicy: "Always"
          volumeMounts:
          - mountPath: /test-pod
            name: test-volume
          env:
          - name: CC_INCLUSTER 1
            value: "true"
          command:
          - "/bin/sh"
          - "-ec"
          - |
            /bin/cluster-capacity --podspec=/test-pod/pod.yaml --verbose
        restartPolicy: "Never"
        serviceAccountName: cluster-capacity-sa
        volumes:
        - name: test-volume
          configMap:
            name: cluster-capacity-configmap
```

1 クラスター容量ツールにクラスター内で Pod として実行されていることを認識させる環境変数です。
ConfigMap の **pod.yaml** キーは **Pod** 仕様ファイル名と同じですが、これは必須ではありません。これを実行することで、入力 Pod 仕様ファイルは **/test-pod/pod.yaml** として Pod 内でアクセスできます。

7. クラスター容量イメージを Pod のジョブとして実行します。

```
$ oc create -f cluster-capacity-job.yaml
```

8. ジョブログを確認し、クラスター内でスケジュールできる Pod の数を確認します。

```
$ oc logs jobs/cluster-capacity-job
```

出力例

```
small-pod pod requirements:
        - CPU: 150m
        - Memory: 100Mi

The cluster can schedule 52 instance(s) of the pod small-pod.

Termination reason: Unschedulable: No nodes are available that match all of the
following predicates:: Insufficient cpu (2).

Pod distribution among nodes:
small-pod
        - 192.168.124.214: 26 instance(s)
        - 192.168.124.120: 26 instance(s)
```

7.3. 制限範囲によるリソース消費の制限

デフォルトで、コンテナーは OpenShift Container Platform クラスターのバインドされていないコンピュートリソースで実行されます。制限範囲については、プロジェクト内の特定オブジェクトのリソースの消費を制限できます。

- Pod およびコンテナー: Pod およびそれらのコンテナーの CPU およびメモリーの最小および最大要件を設定できます。
- イメージストリーム: **ImageStream** オブジェクトのイメージおよびタグの数に制限を設定できます。
- イメージ: 内部レジストリーにプッシュできるイメージのサイズを制限することができます。
- 永続ボリューム要求 (PVC): 要求できる PVC のサイズを制限できます。

Pod が制限範囲で課される制約を満たさない場合、Pod を namespace に作成することはできません。

7.3.1. 制限範囲について

LimitRange オブジェクトで定義される制限範囲。プロジェクトのリソース消費を制限します。プロジェクトで、Pod、コンテナー、イメージ、イメージストリーム、または永続ボリューム要求 (PVC) の特定のリソース制限を設定できます。

すべてのリソース作成および変更要求は、プロジェクトのそれぞれの **LimitRange** オブジェクトに対して評価されます。リソースが列挙される制約のいずれかに違反する場合、そのリソースは拒否されます。

以下は、Pod、コンテナー、イメージ、イメージストリーム、または PVC のすべてのコンポーネントの制限範囲オブジェクトを示しています。同じオブジェクト内のこれらのコンポーネントのいずれかまたはすべての制限を設定できます。リソースを制御するプロジェクトごとに、異なる制限範囲オブジェクトを作成します。

コンテナーの制限オブジェクトのサンプル

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits"
```

```
spec:
  limits:
    - type: "Container"
      max:
        cpu: "2"
        memory: "1Gi"
      min:
        cpu: "100m"
        memory: "4Mi"
      default:
        cpu: "300m"
        memory: "200Mi"
      defaultRequest:
        cpu: "200m"
        memory: "100Mi"
      maxLimitRequestRatio:
        cpu: "10"
```

7.3.1.1. コンポーネントの制限について

以下の例は、それぞれのコンポーネントの制限範囲パラメーターを示しています。これらの例は明確にするために使用されます。必要に応じて、いずれかまたはすべてのコンポーネントの単一の **LimitRange** オブジェクトを作成できます。

7.3.1.1.1. コンテナーの制限

制限範囲により、Pod の各コンテナーが特定のプロジェクトについて要求できる最小および最大 CPU およびメモリーを指定できます。コンテナーがプロジェクトに作成される場合、**Pod** 仕様のコンテナー CPU およびメモリー要求は **LimitRange** オブジェクトに設定される値に準拠する必要があります。そうでない場合には、Pod は作成されません。

- コンテナーの CPU またはメモリーの要求および制限は、**LimitRange** オブジェクトで指定されるコンテナーの **min** リソース制約以上である必要があります。
- コンテナーの CPU またはメモリーの要求と制限は、**LimitRange** オブジェクトで指定されたコンテナーの **max** リソース制約以下である必要があります。
  LimitRange オブジェクトが **max** CPU を定義する場合、 **Pod** 仕様に CPU **request** 値を定義する必要はありません。ただし、制限範囲で指定される最大 CPU 制約を満たす CPU **limit** 値を指定する必要があります。
- コンテナー制限の要求に対する比率は、**LimitRange** オブジェクトに指定されるコンテナーの **maxLimitRequestRatio** 値以下である必要があります。
  LimitRange オブジェクトで **maxLimitRequestRatio** 制約を定義する場合、新規コンテナーには **request** および **limit** 値の両方が必要になります。OpenShift Container Platform は、 **limit** を **request** で除算して制限の要求に対する比率を算出します。この値は、1 より大きい正の整数でなければなりません。

  たとえば、コンテナーの **limit** 値が **cpu: 500** で、**request** 値が **cpu: 100** である場合、**cpu** の要求に対する制限の比は **5** になります。この比率は **maxLimitRequestRatio** より小さいか等しくなければなりません。

Pod 仕様でコンテナーリソースメモリーまたは制限を指定しない場合、制限範囲オブジェクトに指定されるコンテナーの **default** または **defaultRequest** CPU およびメモリー値はコンテナーに割り当てられます。

コンテナー LimitRange オブジェクトの定義

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" 1
spec:
  limits:
    - type: "Container"
      max:
        cpu: "2" 2
        memory: "1Gi" 3
      min:
        cpu: "100m" 4
        memory: "4Mi" 5
      default:
        cpu: "300m" 6
        memory: "200Mi" 7
      defaultRequest:
        cpu: "200m" 8
        memory: "100Mi" 9
      maxLimitRequestRatio:
        cpu: "10" 10
```

1. 制限範囲オブジェクトの名前です。
2. Pod の単一コンテナーが要求できる CPU の最大量です。
3. Pod の単一コンテナーが要求できるメモリーの最大量です。
4. Pod の単一コンテナーが要求できる CPU の最小量です。
5. Pod の単一コンテナーが要求できるメモリーの最小量です。
6. コンテナーが使用できる CPU のデフォルト量 (**Pod** 仕様に指定されていない場合)。
7. コンテナーが使用できるメモリーのデフォルト量 (**Pod** 仕様に指定されていない場合)。
8. コンテナーが要求できる CPU のデフォルト量 (**Pod** 仕様に指定されていない場合)。
9. コンテナーが要求できるメモリーのデフォルト量 (**Pod** 仕様に指定されていない場合)。
10. コンテナーの要求に対する制限の最大比率。

7.3.1.1.2. Pod の制限

制限範囲により、所定プロジェクトの Pod 全体でのすべてのコンテナーの CPU およびメモリーの最小および最大の制限を指定できます。コンテナーをプロジェクトに作成するには、**Pod** 仕様のコンテナー CPU およびメモリー要求は **LimitRange** オブジェクトに設定される値に準拠する必要があります。そうでない場合には、Pod は作成されません。

Pod 仕様でコンテナーリソースメモリーまたは制限を指定しない場合、制限範囲オブジェクトに指定されるコンテナーの **default** または **defaultRequest** CPU およびメモリー値はコンテナーに割り当てられます。

Pod のすべてのコンテナーにおいて、以下を満たしている必要があります。

- コンテナーの CPU またはメモリーの要求および制限は、**LimitRange** オブジェクトに指定される Pod の **min** リソース制約以上である必要があります。
- コンテナーの CPU またはメモリーの要求および制限は、**LimitRange** オブジェクトに指定される Pod の **max** リソース制約以下である必要があります。
- コンテナー制限の要求に対する比率は、**LimitRange** オブジェクトに指定される **maxLimitRequestRatio** 制約以下である必要があります。

Pod LimitRange オブジェクト定義

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" 1
spec:
  limits:
    - type: "Pod"
      max:
        cpu: "2" 2
        memory: "1Gi" 3
      min:
        cpu: "200m" 4
        memory: "6Mi" 5
      maxLimitRequestRatio:
        cpu: "10" 6
```

1. 制限範囲オブジェクトの名前です。
2. すべてのコンテナーにおいて Pod が要求できる CPU の最大量です。
3. すべてのコンテナーにおいて Pod が要求できるメモリーの最大量です。
4. すべてのコンテナーにおいて Pod が要求できる CPU の最小量です。
5. すべてのコンテナーにおいて Pod が要求できるメモリーの最小量です。
6. コンテナーの要求に対する制限の最大比率。

7.3.1.1.3. イメージの制限

LimitRange オブジェクトにより、内部レジストリーにプッシュできるイメージの最大サイズを指定できます。

イメージを内部レジストリーにプッシュする場合には、以下が当てはまります。

- イメージのサイズは、**LimitRange** オブジェクトで指定されるイメージの **max** サイズ以下である必要があります。

イメージ LimitRange オブジェクトの定義

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
```

```
  name: "resource-limits" 1
spec:
  limits:
    - type: openshift.io/Image
      max:
        storage: 1Gi 2
```

1 **LimitRange** オブジェクトの名前。

2 内部レジストリーにプッシュできるイメージの最大サイズ。

注記

制限を超える Blob がレジストリーにアップロードされないようにするために、クォータを実施するようレジストリーを設定する必要があります。

警告

イメージのサイズは、アップロードされるイメージのマニフェストで常に表示される訳ではありません。これは、とりわけ Docker 1.10 以上で作成され、v2 レジストリーにプッシュされたイメージの場合に該当します。このようなイメージが古い Docker デーモンでプルされると、イメージマニフェストはレジストリーによってスキーマ v1 に変換されますが、この場合サイズ情報が欠落します。イメージに設定されるストレージの制限がこのアップロードを防ぐことはありません。

現在、この問題への対応が行われています。

7.3.1.1.4. イメージストリームの制限

LimitRange オブジェクトにより、イメージストリームの制限を指定できます。

各イメージストリームについて、以下が当てはまります。

- **ImageStream** 仕様のイメージタグ数は、**LimitRange** オブジェクトの **openshift.io/image-tags** 制約以下である必要があります。
- **ImageStream** 仕様のイメージへの一意の参照数は、制限範囲オブジェクトの **openshift.io/images** 制約以下である必要があります。

イメージストリーム LimitRange オブジェクト定義

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" 1
spec:
  limits:
    - type: openshift.io/ImageStream
```

```
    max:
      openshift.io/image-tags: 20 (2)
      openshift.io/images: 30 (3)
```

(1) **LimitRange** オブジェクトの名前。

(2) イメージストリーム仕様の **imagestream.spec.tags** パラメーターの一意のイメージタグの最大数。

(3) **imagestream** 仕様の **imagestream.status.tags** パラメーターの一意のイメージ参照の最大数。

openshift.io/image-tags リソースは、一意のイメージ参照を表します。使用できる参照は、**ImageStreamTag**、**ImageStreamImage** および **DockerImage** になります。タグは、**oc tag** および **oc import-image** コマンドを使用して作成できます。内部参照か外部参照であるかの区別はありません。ただし、**ImageStream** の仕様でタグ付けされる一意の参照はそれぞれ 1 回のみカウントされます。内部コンテナーイメージレジストリーへのプッシュを制限しませんが、タグの制限に役立ちます。

openshift.io/images リソースは、イメージストリームのステータスに記録される一意のイメージ名を表します。これにより、内部レジストリーにプッシュできるイメージ数を制限できます。内部参照か外部参照であるかの区別はありません。

7.3.1.1.5. 永続ボリューム要求 (PVC) の制限

LimitRange オブジェクトにより、永続ボリューム要求 (PVC) で要求されるストレージを制限できます。

プロジェクトのすべての永続ボリューム要求 (PVC) において、以下が一致している必要があります。

- 永続ボリューム要求 (PVC) のリソース要求は、**LimitRange** オブジェクトに指定される PVC の **min** 制約以上である必要があります。
- 永続ボリューム要求 (PVC) のリソース要求は、**LimitRange** オブジェクトに指定される PVC の **max** 制約以下である必要があります。

PVC LimitRange オブジェクト定義

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" (1)
spec:
  limits:
    - type: "PersistentVolumeClaim"
      min:
        storage: "2Gi" (2)
      max:
        storage: "50Gi" (3)
```

(1) **LimitRange** オブジェクトの名前。

(2) 永続ボリューム要求 (PVC) で要求できるストレージの最小量です。

(3) 永続ボリューム要求 (PVC) で要求できるストレージの最大量です。

7.3.2. 制限範囲の作成

制限範囲をプロジェクトに適用するには、以下を実行します。

1. 必要な仕様で **LimitRange** オブジェクトを作成します。

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" (1)
spec:
  limits:
    - type: "Pod" (2)
      max:
        cpu: "2"
        memory: "1Gi"
      min:
        cpu: "200m"
        memory: "6Mi"
    - type: "Container" (3)
      max:
        cpu: "2"
        memory: "1Gi"
      min:
        cpu: "100m"
        memory: "4Mi"
      default: (4)
        cpu: "300m"
        memory: "200Mi"
      defaultRequest: (5)
        cpu: "200m"
        memory: "100Mi"
      maxLimitRequestRatio: (6)
        cpu: "10"
    - type: openshift.io/Image (7)
      max:
        storage: 1Gi
    - type: openshift.io/ImageStream (8)
      max:
        openshift.io/image-tags: 20
        openshift.io/images: 30
    - type: "PersistentVolumeClaim" (9)
      min:
        storage: "2Gi"
      max:
        storage: "50Gi"
```

(1) **LimitRange** オブジェクトの名前を指定します。

(2) Pod の制限を設定するには、必要に応じて CPU およびメモリーの最小および最大要求を指定します。

(3) コンテナーの制限を設定するには、必要に応じて CPU およびメモリーの最小および最大要求を指定します。

4. オプション:コンテナーの場合、**Pod** 仕様で指定されていない場合、コンテナーが使用できる CPU またはメモリーのデフォルト量を指定します。

5. オプション:コンテナーの場合、**Pod** 仕様で指定されていない場合、コンテナーが要求できる CPU またはメモリーのデフォルト量を指定します。

6. オプション:コンテナーの場合、**Pod** 仕様で指定できる要求に対する制限の最大比率を指定します。

7. Image オブジェクトの制限を設定するには、内部レジストリーにプッシュできるイメージの最大サイズを設定します。

8. イメージストリームの制限を設定するには、必要に応じて **ImageStream** オブジェクトファイルにあるイメージタグおよび参照の最大数を設定します。

9. 永続ボリューム要求 (PVC) の制限を設定するには、要求できるストレージの最小および最大量を設定します。

2. オブジェクトを作成します。

```
$ oc create -f <limit_range_file> -n <project> 1
```

1. 作成した YAML ファイルの名前と、制限を適用する必要のあるプロジェクトを指定します。

7.3.3. 制限の表示

Web コンソールでプロジェクトの **Quota** ページに移動し、プロジェクトで定義される制限を表示できます。

CLI を使用して制限範囲の詳細を表示することもできます。

1. プロジェクトで定義される **LimitRange** オブジェクトの一覧を取得します。たとえば、**demoproject** というプロジェクトの場合は以下のようになります。

```
$ oc get limits -n demoproject
```

```
NAME              CREATED AT
resource-limits   2020-07-15T17:14:23Z
```

2. 関連のある **LimitRange** オブジェクトを記述します。 たとえば、**resource-limits** 制限範囲の場合は以下のようになります。

```
$ oc describe limits resource-limits -n demoproject
```

```
Name:                   resource-limits
Namespace:              demoproject
Type                    Resource            Min     Max     Default Request Default Limit   Max
Limit/Request Ratio
----                    --------            ---     ---     --------------- -------------   -----------------------
Pod                     cpu                 200m    2       -               -               -
Pod                     memory              6Mi     1Gi     -               -               -
```

```
Container                cpu                100m    2       200m          300m          10
Container                memory             4Mi     1Gi     100Mi         200Mi         -
openshift.io/Image       storage            -       1Gi     -             -             -
openshift.io/ImageStream openshift.io/image -       12      -             -             -
openshift.io/ImageStream openshift.io/image-tags -       10      -             -             -
PersistentVolumeClaim    storage            -       50Gi    -             -             -
```

7.3.4. 制限範囲の削除

プロジェクトで制限を実施しないように有効な **LimitRange** オブジェクト削除するには、以下を実行します。

1. 以下のコマンドを実行します。

```
$ oc delete limits <limit_name>
```

7.4. コンテナーメモリーとリスク要件を満たすためのクラスターメモリーの設定

クラスター管理者は、以下を実行し、クラスターがアプリケーションメモリーの管理を通じて効率的に動作するようにすることができます。

- コンテナー化されたアプリケーションコンポーネントのメモリーおよびリスク要件を判別し、それらの要件を満たすようコンテナーメモリーパラメーターを設定する
- コンテナー化されたアプリケーションランタイム (OpenJDK など) を、設定されたコンテナーメモリーパラメーターに基づいて最適に実行されるよう設定する
- コンテナーでの実行に関連するメモリー関連のエラー状態を診断し、これを解決する

7.4.1. アプリケーションメモリーの管理について

まず OpenShift Container Platform によるコンピュートリソースの管理方法の概要をよく読んでから次の手順に進むことをお勧めします。

各種のリソース (メモリー、cpu、ストレージ) に応じて、OpenShift Container Platform ではオプションの **要求** および **制限** の値を Pod の各コンテナーに設定できます。

メモリー要求とメモリー制限について、以下の点に注意してください。

- **メモリー要求**
  - メモリー要求値は、指定される場合 OpenShift Container Platform スケジューラーに影響を与えます。スケジューラーは、コンテナーのノードへのスケジュール時にメモリー要求を考慮し、コンテナーの使用のために選択されたノードで要求されたメモリーをフェンスオフします。
  - ノードのメモリーが使い切られると、OpenShift Container Platform はメモリー使用がメモリー要求を最も超過しているコンテナーのエビクションを優先します。メモリー消費の深刻な状況が生じる場合、ノードの OOM killer は同様のメトリクスに基づいてコンテナーでプロセスを選択し、これを強制終了する場合があります。
  - クラスター管理者は、メモリー要求値に対してクォータを割り当てるか、デフォルト値を割り当てることができます。

- クラスター管理者は、クラスターのオーバーコミットを管理するために開発者が指定するメモリー要求の値を上書きできます。

- **メモリー制限**
  - メモリー制限値が指定されている場合、コンテナーのすべてのプロセスに割り当て可能なメモリーにハード制限を指定します。
  - コンテナーのすべてのプロセスで割り当てられるメモリーがメモリー制限を超過する場合、ノードの OOM (Out of Memory) killer はコンテナーのプロセスをすぐに選択し、これを強制終了します。
  - メモリー要求とメモリー制限の両方が指定される場合、メモリー制限の値はメモリー要求の値よりも大きいか、またはこれと等しくなければなりません。
  - クラスター管理者は、メモリーの制限値に対してクォータを割り当てるか、デフォルト値を割り当てることができます。
  - 最小メモリー制限は 12 MB です。**Cannot allocate memory** Pod イベントのためにコンテナーの起動に失敗すると、メモリー制限は低くなります。メモリー制限を引き上げるか、またはこれを削除します。制限を削除すると、Pod は制限のないノードのリソースを消費できるようになります。

7.4.1.1. アプリケーションメモリーストラテジーの管理

OpenShift Container Platform でアプリケーションメモリーをサイジングする手順は以下の通りです。

1. **予想されるコンテナーのメモリー使用の判別**
   必要時に予想される平均およびピーク時のコンテナーのメモリー使用を判別します (例: 別の負荷テストを実行)。コンテナーで並行して実行されている可能性のあるすべてのプロセスを必ず考慮に入れるようにしてください。 たとえば、メインのアプリケーションは付属スクリプトを生成しているかどうかを確認します。

2. **リスク選好 (risk appetite) の判別**
   エビクションのリスク選好を判別します。リスク選好のレベルが低い場合、コンテナーは予想されるピーク時の使用量と安全マージンのパーセンテージに応じてメモリーを要求します。リスク選好が高くなる場合、予想される平均の使用量に応じてメモリーを要求することがより適切な場合があります。

3. **コンテナーのメモリー要求の設定**
   上記に基づいてコンテナーのメモリー要求を設定します。要求がアプリケーションのメモリー使用をより正確に表示することが望ましいと言えます。要求が高すぎる場合には、クラスターおよびクォータの使用が非効率となります。要求が低すぎる場合、アプリケーションのエビクションの可能性が高くなります。

4. **コンテナーのメモリー制限の設定 (必要な場合)**
   必要時にコンテナーのメモリー制限を設定します。制限を設定すると、コンテナーのすべてのプロセスのメモリー使用量の合計が制限を超える場合にコンテナーのプロセスがすぐに強制終了されるため、いくつかの利点をもたらします。まずは予期しないメモリー使用の超過を早期に明確にする (fail fast (早く失敗する)) ことができ、次にプロセスをすぐに中止できます。

   一部の OpenShift Container Platform クラスターでは制限値を設定する必要があります。 制限に基づいて要求を上書きする場合があります。 また、一部のアプリケーションイメージは、要求値よりも検出が簡単なことから設定される制限値に依存します。

   メモリー制限が設定される場合、これは予想されるピーク時のコンテナーのメモリー使用量と安全マージンのパーセンテージよりも低い値に設定することはできません。

5. **アプリケーションが調整されていることの確認**
   適切な場合は、設定される要求および制限値に関連してアプリケーションが調整されていることを確認します。この手順は、JVM などのメモリーをプールするアプリケーションにおいてとくに当てはまります。残りの部分では、これについて説明します。

関連情報

- コンピュートリソースとコンテナーについて

7.4.2. OpenShift Container Platform の OpenJDK 設定について

デフォルトの OpenJDK 設定はコンテナー化された環境では機能しません。そのため、コンテナーで OpenJDK を実行する場合は常に追加の Java メモリー設定を指定する必要があります。

JVM のメモリーレイアウトは複雑で、バージョンに依存しており、本書ではこれについて詳細には説明しません。ただし、コンテナーで OpenJDK を実行する際のスタートにあたって少なくとも以下の 3 つのメモリー関連のタスクが主なタスクになります。

1. JVM 最大ヒープサイズを上書きする。
2. JVM が未使用メモリーをオペレーティングシステムに解放するよう促す (適切な場合)。
3. コンテナー内のすべての JVM プロセスが適切に設定されていることを確認する。

コンテナーでの実行に向けて JVM ワークロードを最適に調整する方法については本書では扱いませんが、これには複数の JVM オプションを追加で設定することが必要になる場合があります。

7.4.2.1. JVM の最大ヒープサイズを上書きする方法について

数多くの Java ワークロードにおいて、JVM ヒープはメモリーの最大かつ単一のコンシューマーです。現時点で OpenJDK は、OpenJDK がコンテナー内で実行されているかにかかわらず、ヒープに使用されるコンピュートノードのメモリーの最大 1/4 (1/**-XX:MaxRAMFraction**) を許可するようデフォルトで設定されます。そのため、コンテナーのメモリー制限も設定されている場合には、この動作をオーバーライドすることが **必須** です。

上記を実行する方法として、2 つ以上の方法を使用できます:

1. コンテナーのメモリー制限が設定されており、JVM で実験的なオプションがサポートされている場合には、**-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap** を設定します。

   注記

   JDK 11 では **UseCGroupMemoryLimitForHeap** オプションが削除されました。**-XX:+UseContainerSupport** を代わりに使用します。

   これにより、**-XX:MaxRAM** がコンテナーのメモリー制限に設定され、最大ヒープサイズ (**-XX:MaxHeapSize** / **-Xmx**) が 1/**-XX:MaxRAMFraction** に設定されます (デフォルトでは 1/4)。

2. **-XX:MaxRAM**、**-XX:MaxHeapSize** または **-Xmx** のいずれかを直接上書きします。
   このオプションには、値のハードコーディングが必要になりますが、安全マージンを計算できるという利点があります。

7.4.2.2. JVM で未使用メモリーをオペレーティングシステムに解放するよう促す方法について

デフォルトで、OpenJDK は未使用メモリーをオペレーティングシステムに積極的に返しません。これは多くのコンテナー化された Java ワークロードには適していますが、例外として、コンテナー内に JVM と共存する追加のアクティブなプロセスがあるワークロードの場合を考慮する必要があります。それらの追加のプロセスはネイティブのプロセスである場合や追加の JVM の場合、またはこれら 2 つの組み合わせである場合もあります。

OpenShift Container Platform Jenkins maven スレーブイメージは、以下の JVM 引数を使用して JVM に未使用メモリーをオペレーティングシステムに解放するよう促します。

```
-XX:+UseParallelGC
-XX:MinHeapFreeRatio=5 -XX:MaxHeapFreeRatio=10 -XX:GCTimeRatio=4
-XX:AdaptiveSizePolicyWeight=90.
```

これらの引数は、割り当てられたメモリーが使用中のメモリー (**-XX:MaxHeapFreeRatio**) の 110% を超え、ガベージコレクター (**-XX:GCTimeRatio**) での CPU 時間の 20% を使用する場合は常にヒープメモリーをオペレーティングシステムに返すことが意図されています。アプリケーションのヒープ割り当てが初期のヒープ割り当て (**-XX:InitialHeapSize** / **-Xms** で上書きされる) を下回ることはありません。詳細情報については、Tuning Java's footprint in OpenShift (Part 1) 、Tuning Java's footprint in OpenShift (Part 2)、および OpenJDK and Containers を参照してください。

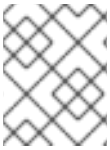
7.4.2.3. コンテナー内のすべての JVM プロセスが適切に設定されていることを確認する方法について

複数の JVM が同じコンテナーで実行される場合、それらすべてが適切に設定されていることを確認する必要があります。多くのワークロードでは、それぞれの JVM に memory budget のパーセンテージを付与する必要があります。 これにより大きな安全マージンが残される場合があります。

多くの Java ツールは JVM を設定するために各種の異なる環境変数 (**JAVA_OPTS**、**GRADLE_OPTS**、**MAVEN_OPTS** など) を使用します。 適切な設定が適切な JVM に渡されていることを確認するのが容易でない場合もあります。

JAVA_TOOL_OPTIONS 環境変数は常に OpenJDK によって考慮され、**JAVA_TOOL_OPTIONS** に指定された値は、JVM コマンドラインに指定される他のオプションによって上書きされます。デフォルトでは、これらのオプションがスレーブイメージで実行されるすべての JVM ワークロードに対してデフォルトで使用されていることを確認するには、OpenShift Container Platform Jenkins maven スレーブイメージを以下のように設定します。

```
JAVA_TOOL_OPTIONS="-XX:+UnlockExperimentalVMOptions
-XX:+UseCGroupMemoryLimitForHeap -Dsun.zip.disableMemoryMapping=true"
```



注記

JDK 11 では **UseCGroupMemoryLimitForHeap** オプションが削除されました。**-XX:+UseContainerSupport** を代わりに使用します。

この設定は、追加オプションが要求されないことを保証する訳ではなく、有用な開始点になることを意図しています。

7.4.3. Pod 内でのメモリー要求および制限の検索

Pod 内からメモリー要求および制限を動的に検出するアプリケーションでは Downward API を使用する必要があります。

手順

1. **MEMORY_REQUEST** と **MEMORY_LIMIT** スタンザを追加するように Pod を設定します。

```
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  containers:
  - name: test
    image: fedora:latest
    command:
    - sleep
    - "3600"
    env:
    - name: MEMORY_REQUEST (1)
      valueFrom:
        resourceFieldRef:
          containerName: test
          resource: requests.memory
    - name: MEMORY_LIMIT (2)
      valueFrom:
        resourceFieldRef:
          containerName: test
          resource: limits.memory
    resources:
      requests:
        memory: 384Mi
      limits:
        memory: 512Mi
```

(1) このスタンザを追加して、アプリケーションメモリーの要求値を見つけます。

(2) このスタンザを追加して、アプリケーションメモリーの制限値を見つけます。

2. Pod を作成します。

```
$ oc create -f <file-name>.yaml
```

3. リモートシェルを使用して Pod にアクセスします。

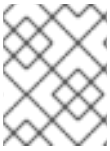
```
$ oc rsh test
```

4. 要求された値が適用されていることを確認します。

```
$ env | grep MEMORY | sort
```

出力例

```
MEMORY_LIMIT=536870912
MEMORY_REQUEST=402653184
```



注記

メモリー制限値は、**/sys/fs/cgroup/memory/memory.limit_in_bytes** ファイルによってコンテナー内から読み取ることもできます。

7.4.4. OOM の強制終了ポリシーについて

OpenShift Container Platform は、コンテナーのすべてのプロセスのメモリー使用量の合計がメモリー制限を超えるか、またはノードのメモリーを使い切られるなどの深刻な状態が生じる場合にコンテナーのプロセスを強制終了できます。

プロセスが OOM (Out of Memory) によって強制終了される場合、コンテナーがすぐに終了する場合があります。コンテナーの PID 1 プロセスが **SIGKILL** を受信する場合、コンテナーはすぐに終了します。それ以外の場合、コンテナーの動作は他のプロセスの動作に依存します。

たとえば、コンテナーのプロセスは、SIGKILL シグナルを受信したことを示すコード 137 で終了します。

コンテナーがすぐに終了しない場合、OOM による強制終了は以下のように検出できます。

1. リモートシェルを使用して Pod にアクセスします。

```
# oc rsh test
```

2. 以下のコマンドを実行して、**/sys/fs/cgroup/memory/memory.oom_control** で現在の OOM kill カウントを表示します。

```
$ grep '^oom_kill ' /sys/fs/cgroup/memory/memory.oom_control
oom_kill 0
```

3. 以下のコマンドを実行して、Out Of Memory (OOM) による強制終了を促します。

```
$ sed -e '' </dev/zero
```

出力例

```
Killed
```

4. 以下のコマンドを実行して、**sed** コマンドの終了ステータスを表示します。

```
$ echo $?
```

出力例

```
137
```

137 コードは、コンテナーのプロセスが、SIGKILL シグナルを受信したことを示すコード 137 で終了していることを示唆します。

5. 以下のコマンドを実行して、**/sys/fs/cgroup/memory/memory.oom_control** の OOM kill カウンターの増分を表示します。

```
$ grep '^oom_kill ' /sys/fs/cgroup/memory/memory.oom_control
oom_kill 1
```

Pod の 1 つ以上のプロセスが OOM で強制終了され、Pod がこれに続いて終了する場合 (即時であるかどうかは問わない)、フェーズは **Failed**、理由は **OOMKilled** になります。OOM で強制終了された Pod は **restartPolicy** の値によって再起動する場合があります。再起動されない場合は、レプリケーションコントローラーなどのコントローラーが Pod の失敗したステータスを認識し、古い Pod に置き換わる新規 Pod を作成します。

以下のコマンドを使用して Pod のステータスを取得します。

```
$ oc get pod test
```

出力例

```
NAME      READY     STATUS      RESTARTS   AGE
test      0/1       OOMKilled   0          1m
```

- Pod が再起動されていない場合は、以下のコマンドを実行して Pod を表示します。

  ```
  $ oc get pod test -o yaml
  ```

  出力例

  ```
  ...
  status:
    containerStatuses:
    - name: test
      ready: false
      restartCount: 0
      state:
        terminated:
          exitCode: 137
          reason: OOMKilled
    phase: Failed
  ```

- 再起動した場合は、以下のコマンドを実行して Pod を表示します。

  ```
  $ oc get pod test -o yaml
  ```

  出力例

  ```
  ...
  status:
    containerStatuses:
    - name: test
      ready: true
      restartCount: 1
      lastState:
        terminated:
          exitCode: 137
          reason: OOMKilled
  ```

```
    state:
      running:
  phase: Running
```

7.4.5. Pod エビクションについて

OpenShift Container Platform は、ノードのメモリーが使い切られると、そのノードから Pod をエビクトする場合があります。メモリー消費の度合いによって、エビクションは正常に行われる場合もあれば、そうでない場合もあります。正常なエビクションは、各コンテナーのメインプロセス (PID 1) が SIGTERM シグナルを受信してから、プロセスがすでに終了していない場合は後になって SIGKILL シグナルを受信することを意味します。正常ではないエビクションは各コンテナーのメインプロセスが SIGKILL シグナルを即時に受信することを示します。

エビクトされた Pod のフェーズは **Failed** になり、理由 は **Evicted** になります。この場合、**restartPolicy** の値に関係なく再起動されません。ただし、レプリケーションコントローラーなどのコントローラーは Pod の失敗したステータスを認識し、古い Pod に置き換わる新規 Pod を作成します。

```
$ oc get pod test
```

出力例

```
NAME      READY     STATUS    RESTARTS   AGE
test      0/1       Evicted   0          1m
```

```
$ oc get pod test -o yaml
```

出力例

```
...
status:
  message: 'Pod The node was low on resource: [MemoryPressure].'
  phase: Failed
  reason: Evicted
```

7.5. オーバーコミットされたノード上に POD を配置するためのクラスターの設定

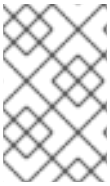
オーバーコミット とは、コンテナーの計算リソース要求と制限の合計が、そのシステムで利用できるリソースを超えた状態のことです。オーバーコミットの使用は、容量に対して保証されたパフォーマンスのトレードオフが許容可能である開発環境において必要になる場合があります。

コンテナーは、コンピュートリソース要求および制限を指定することができます。要求はコンテナーのスケジューリングに使用され、最小限のサービス保証を提供します。制限は、ノード上で消費できるコンピュートリソースの量を制限します。

スケジューラーは、クラスター内のすべてのノードにおけるコンピュートリソース使用の最適化を試行します。これは Pod のコンピュートリソース要求とノードの利用可能な容量を考慮に入れて Pod を特定のノードに配置します。

OpenShift Container Platform 管理者は、オーバーコミットのレベルを制御し、ノード上のコンテナーの密度を管理できるようになりました。クラスターレベルのオーバーコミットを

ClusterResourceOverride Operator を使用して設定し、開発者用のコンテナーに設定された要求と制限の比率について上書きすることができます。ノードのオーバーコミット、および プロジェクトのメモリーおよび CPU 制限およびデフォルト と共に、オーバーコミットの必要なレベルを実現するためにリソース制限および要求を調整することができます。



注記

OpenShift Container Platform では、クラスターレベルのオーバーコミットを有効にする必要があります。ノードのオーバーコミットはデフォルトで有効にされています。ノードのオーバーコミットの無効化 を参照してください。

7.5.1. リソース要求とオーバーコミット

各コンピュートリソースについて、コンテナーはリソース要求および制限を指定できます。スケジューリングの決定は要求に基づいて行われ、ノードに要求される値を満たす十分な容量があることが確認されます。コンテナーが制限を指定するものの、要求を省略する場合、要求はデフォルトで制限値に設定されます。コンテナーは、ノードの指定される制限を超えることはできません。

制限の実施方法は、コンピュートリソースのタイプによって異なります。コンテナーが要求または制限を指定しない場合、コンテナーはリソース保証のない状態でノードにスケジュールされます。実際に、コンテナーはローカルの最も低い優先順位で利用できる指定リソースを消費できます。リソースが不足する状態では、リソース要求を指定しないコンテナーに最低レベルの QoS (Quality of Service) が設定されます。

スケジューリングは要求されるリソースに基づいて行われる一方で、クォータおよびハード制限はリソース制限のことを指しており、これは要求されるリソースよりも高い値に設定できます。要求と制限の間の差異は、オーバーコミットのレベルを定めるものとなります。 たとえば、コンテナーに 1Gi のメモリー要求と 2Gi のメモリー制限が指定される場合、コンテナーのスケジューリングはノードで 1Gi を利用可能とする要求に基づいて行われますが、 2Gi まで使用することができます。 そのため、この場合のオーバーコミットは 200% になります。

7.5.2. Cluster Resource Override Operator を使用したクラスターレベルのオーバーコミット

Cluster Resource Override Operator は、クラスター内のすべてのノードでオーバーコミットのレベルを制御し、コンテナーの密度を管理できる受付 Webhook です。Operator は、特定のプロジェクトのノードが定義されたメモリーおよび CPU 制限を超える場合について制御します。

以下のセクションで説明されているように、OpenShift Container Platform コンソールまたは CLI を使用して Cluster Resource Override Operator をインストールする必要があります。インストール時に、以下の例のように、オーバーコミットのレベルを設定する **ClusterResourceOverride** カスタムリソース (CR) を作成します。

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
  name: cluster (1)
spec:
  podResourceOverride:
    spec:
      memoryRequestToLimitPercent: 50 (2)
      cpuRequestToLimitPercent: 25 (3)
      limitCPUToMemoryPercent: 200 (4)
```

1. 名前は **cluster** でなければなりません。
2. オプション:コンテナーのメモリー制限が指定されているか、またはデフォルトに設定されている場合、メモリー要求は制限のパーセンテージ (1-100) に対して上書きされます。デフォルトは 50 です。
3. オプション:コンテナーの CPU 制限が指定されているか、またはデフォルトに設定されている場合、CPU 要求は、1-100 までの制限のパーセンテージに対応して上書きされます。デフォルトは 25 です。
4. オプション:コンテナーのメモリー制限が指定されているか、デフォルトに設定されている場合、CPU 制限は、指定されている場合にメモリーのパーセンテージに対して上書きされます。1Gi の RAM の 100 パーセントでのスケーリングは、1 CPU コアに等しくなります。これは、CPU 要求を上書きする前に処理されます (設定されている場合)。デフォルトは 200 です。

注記

Cluster Resource Override Operator の上書きは、制限がコンテナーに設定されていない場合は影響を与えません。個別プロジェクトごとのデフォルト制限を使用して **LimitRange** オブジェクトを作成するか、または **Pod** 仕様で制限を設定し、上書きが適用されるようにします。

設定時に、以下のラベルを各プロジェクトの namespace オブジェクトに適用し、上書きをプロジェクトごとに有効にできます。

```
apiVersion: v1
kind: Namespace
metadata:

....

  labels:
    clusterresourceoverrides.admission.autoscaling.openshift.io/enabled: "true"

....
```

Operator は **ClusterResourceOverride** CR の有無を監視し、**ClusterResourceOverride** 受付 Webhook が Operator と同じ namespace にインストールされるようにします。

7.5.2.1. Web コンソールを使用した Cluster Resource Override Operator のインストール

クラスターでオーバーコミットを制御できるように、OpenShift Container Platform Web コンソールを使用して Cluster Resource Override Operator をインストールできます。

前提条件

- 制限がコンテナーに設定されていない場合、Cluster Resource Override Operator は影響を与えません。**LimitRange** オブジェクトを使用してプロジェクトのデフォルト制限を指定するか、または **Pod** 仕様で制限を設定して上書きが適用されるようにする必要があります。

手順

OpenShift Container Platform Web コンソールを使って Cluster Resource Override Operator をインストールするには、以下を実行します。

1. OpenShift Container Platform Web コンソールで、**Home → Projects** に移動します。
   a. **Create Project** をクリックします。
   b. **clusterresourceoverride-operator** をプロジェクトの名前として指定します。
   c. **Create** をクリックします。
2. **Operators → OperatorHub** に移動します。
   a. 利用可能な Operator の一覧から **ClusterResourceOverride Operator** を選択し、**Install** をクリックします。
   b. **Install Operator** ページで、**A specific Namespace on the cluster**が **Installation Mode** について選択されていることを確認します。
   c. **clusterresourceoverride-operator** が **Installed Namespace** について選択されていることを確認します。
   d. **Update Channel** および **Approval Strategy** を選択します。
   e. **Install** をクリックします。
3. **Installed Operators** ページで、**ClusterResourceOverride** をクリックします。
   a. **ClusterResourceOverride Operator** の詳細ページで、**Create Instance** をクリックします。
   b. **Create ClusterResourceOverride** ページで、YAML テンプレートを編集して、必要に応じてオーバーコミット値を設定します。

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
  name: cluster (1)
spec:
  podResourceOverride:
    spec:
      memoryRequestToLimitPercent: 50 (2)
      cpuRequestToLimitPercent: 25 (3)
      limitCPUToMemoryPercent: 200 (4)
```

(1) 名前は **cluster** でなければなりません。

(2) オプション:コンテナーメモリーの制限を上書きするためのパーセンテージが使用される場合は、これを 1-100 までの値で指定します。デフォルトは 50 です。

(3) オプション:コンテナー CPU の制限を上書きするためのパーセンテージが使用される場合は、これを 1-100 までの値で指定します。デフォルトは 25 です。

(4) オプション:コンテナーメモリーの制限を上書きするためのパーセンテージが使用される場合は、これを指定します。1Gi の RAM の 100 パーセントでのスケーリングは、1 CPU コアに等しくなります。これは、CPU 要求を上書きする前に処理されます (設定されている場合)。デフォルトは 200 です。

   c. **Create** をクリックします。

4. クラスターカスタムリソースのステータスをチェックして、受付 Webhook の現在の状態を確認します。

   a. **ClusterResourceOverride Operator** ページで、**cluster** をクリックします。

   b. **ClusterResourceOverride Details** ページで、 **YAML** をクリックします。Webhook の呼び出し時に、**mutatingWebhookConfigurationRef** セクションが表示されます。

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"operator.autoscaling.openshift.io/v1","kind":"ClusterResourceOverride","metadata":{"annotations":{},"name":"cluster"},"spec":{"podResourceOverride":{"spec":{"cpuRequestToLimitPercent":25,"limitCPUToMemoryPercent":200,"memoryRequestToLimitPercent":50}}}}
  creationTimestamp: "2019-12-18T22:35:02Z"
  generation: 1
  name: cluster
  resourceVersion: "127622"
  selfLink: /apis/operator.autoscaling.openshift.io/v1/clusterresourceoverrides/cluster
  uid: 978fc959-1717-4bd1-97d0-ae00ee111e8d
spec:
  podResourceOverride:
    spec:
      cpuRequestToLimitPercent: 25
      limitCPUToMemoryPercent: 200
      memoryRequestToLimitPercent: 50
status:

....

    mutatingWebhookConfigurationRef: (1)
      apiVersion: admissionregistration.k8s.io/v1beta1
      kind: MutatingWebhookConfiguration
      name: clusterresourceoverrides.admission.autoscaling.openshift.io
      resourceVersion: "127621"
      uid: 98b3b8ae-d5ce-462b-8ab5-a729ea8f38f3

....
```

(1) **ClusterResourceOverride** 受付 Webhook への参照。

7.5.2.2. CLI を使用した Cluster Resource Override Operator のインストール

OpenShift Container Platform CLI を使用して Cluster Resource Override Operator をインストールし、クラスターでのオーバーコミットを制御できます。

前提条件

- 制限がコンテナーに設定されていない場合、Cluster Resource Override Operator は影響を与えません。**LimitRange** オブジェクトを使用してプロジェクトのデフォルト制限を指定するか、または **Pod** 仕様で制限を設定して上書きが適用されるようにする必要があります。

手順

CLI を使用して Cluster Resource Override Operator をインストールするには、以下を実行します。

1. Cluster Resource Override の namespace を作成します。
   a. Cluster Resource Override Operator の **Namespace** オブジェクト YAML ファイル (**cro-namespace.yaml** など) を作成します。

```
apiVersion: v1
kind: Namespace
metadata:
  name: clusterresourceoverride-operator
```

   b. namespace を作成します。

```
$ oc create -f <file-name>.yaml
```

   以下に例を示します。

```
$ oc create -f cro-namespace.yaml
```

2. Operator グループを作成します。
   a. Cluster Resource Override Operator の **OperatorGroup** オブジェクトの YAML ファイル (cro-og.yaml など) を作成します。

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: clusterresourceoverride-operator
  namespace: clusterresourceoverride-operator
spec:
  targetNamespaces:
    - clusterresourceoverride-operator
```

   b. Operator グループを作成します。

```
$ oc create -f <file-name>.yaml
```

   以下に例を示します。

```
$ oc create -f cro-og.yaml
```

3. サブスクリプションを作成します。
   a. Cluster Resource Override Operator の **Subscription** オブジェクト YAML ファイル (cro-sub.yaml など) を作成します。

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: clusterresourceoverride
  namespace: clusterresourceoverride-operator
```

```
spec:
  channel: "4.7"
  name: clusterresourceoverride
  source: redhat-operators
  sourceNamespace: openshift-marketplace
```

b. サブスクリプションを作成します。

```
$ oc create -f <file-name>.yaml
```

以下に例を示します。

```
$ oc create -f cro-sub.yaml
```

4. **ClusterResourceOverride** カスタムリソース (CR) オブジェクトを **clusterresourceoverride-operator** namespace に作成します。

a. **clusterresourceoverride-operator** namespace に切り替えます。

```
$ oc project clusterresourceoverride-operator
```

b. Cluster Resource Override Operator の **ClusterResourceOverride** オブジェクト YAML ファイル (cro-cr.yaml など) を作成します。

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
   name: cluster (1)
spec:
  podResourceOverride:
    spec:
      memoryRequestToLimitPercent: 50 (2)
      cpuRequestToLimitPercent: 25 (3)
      limitCPUToMemoryPercent: 200 (4)
```

(1) 名前は **cluster** でなければなりません。

(2) オプション:コンテナーメモリーの制限を上書きするためのパーセンテージが使用される場合は、これを 1-100 までの値で指定します。デフォルトは 50 です。

(3) オプション:コンテナー CPU の制限を上書きするためのパーセンテージが使用される場合は、これを 1-100 までの値で指定します。デフォルトは 25 です。

(4) オプション:コンテナーメモリーの制限を上書きするためのパーセンテージが使用される場合は、これを指定します。1Gi の RAM の 100 パーセントでのスケーリングは、1 CPU コアに等しくなります。これは、CPU 要求を上書きする前に処理されます (設定されている場合)。デフォルトは 200 です。

c. **ClusterResourceOverride** オブジェクトを作成します。

```
$ oc create -f <file-name>.yaml
```

以下に例を示します。

```
$ oc create -f cro-cr.yaml
```

5. クラスターカスタムリソースのステータスをチェックして、受付 Webhook の現在の状態を確認します。

```
$ oc get clusterresourceoverride cluster -n clusterresourceoverride-operator -o yaml
```

Webhook の呼び出し時に、**mutatingWebhookConfigurationRef** セクションが表示されます。

出力例

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"operator.autoscaling.openshift.io/v1","kind":"ClusterResourceOverride","metadata":{"annotations":{},"name":"cluster"},"spec":{"podResourceOverride":{"spec":{"cpuRequestToLimitPercent":25,"limitCPUToMemoryPercent":200,"memoryRequestToLimitPercent":50}}}}
  creationTimestamp: "2019-12-18T22:35:02Z"
  generation: 1
  name: cluster
  resourceVersion: "127622"
  selfLink: /apis/operator.autoscaling.openshift.io/v1/clusterresourceoverrides/cluster
  uid: 978fc959-1717-4bd1-97d0-ae00ee111e8d
spec:
  podResourceOverride:
    spec:
      cpuRequestToLimitPercent: 25
      limitCPUToMemoryPercent: 200
      memoryRequestToLimitPercent: 50
status:

....

    mutatingWebhookConfigurationRef: 1
      apiVersion: admissionregistration.k8s.io/v1beta1
      kind: MutatingWebhookConfiguration
      name: clusterresourceoverrides.admission.autoscaling.openshift.io
      resourceVersion: "127621"
      uid: 98b3b8ae-d5ce-462b-8ab5-a729ea8f38f3

....
```

1 **ClusterResourceOverride** 受付 Webhook への参照。

7.5.2.3. クラスターレベルのオーバーコミットの設定

Cluster Resource Override Operator には、Operator がオーバーコミットを制御する必要のある各プロジェクトの **ClusterResourceOverride** カスタムリソース (CR) およびラベルが必要です。

前提条件

- 制限がコンテナーに設定されていない場合、Cluster Resource Override Operator は影響を与えません。**LimitRange** オブジェクトを使用してプロジェクトのデフォルト制限を指定するか、または **Pod** 仕様で制限を設定して上書きが適用されるようにする必要があります。

手順

クラスターレベルのオーバーコミットを変更するには、以下を実行します。

1. **ClusterResourceOverride** CR を編集します。

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
   name: cluster
spec:
  podResourceOverride:
    spec:
      memoryRequestToLimitPercent: 50 (1)
      cpuRequestToLimitPercent: 25 (2)
      limitCPUToMemoryPercent: 200 (3)
```

(1) オプション:コンテナーメモリーの制限を上書きするためのパーセンテージが使用される場合は、これを 1-100 までの値で指定します。デフォルトは 50 です。

(2) オプション:コンテナー CPU の制限を上書きするためのパーセンテージが使用される場合は、これを 1-100 までの値で指定します。デフォルトは 25 です。

(3) オプション:コンテナーメモリーの制限を上書きするためのパーセンテージが使用される場合は、これを指定します。1Gi の RAM の 100 パーセントでのスケーリングは、1 CPU コアに等しくなります。これは、CPU 要求を上書きする前に処理されます (設定されている場合)。デフォルトは 200 です。

2. 以下のラベルが Cluster Resource Override Operator がオーバーコミットを制御する必要のある各プロジェクトの namespace オブジェクトに追加されていることを確認します。

```
apiVersion: v1
kind: Namespace
metadata:

 ...

 labels:
   clusterresourceoverrides.admission.autoscaling.openshift.io/enabled: "true" (1)

...
```

(1) このラベルを各プロジェクトに追加します。

7.5.3. ノードレベルのオーバーコミット

QoS (Quality of Service) 保証、CPU 制限、またはリソースの予約など、特定ノードでオーバーコミットを制御するさまざまな方法を使用できます。特定のノードおよび特定のプロジェクトのオーバーコミットを無効にすることもできます。

7.5.3.1. コンピュートリソースとコンテナーについて

コンピュートリソースについてのノードで実施される動作は、リソースタイプによって異なります。

7.5.3.1.1. コンテナーの CPU 要求について

コンテナーには要求する CPU の量が保証され、さらにコンテナーで指定される任意の制限までノードで利用可能な CPU を消費できます。複数のコンテナーが追加の CPU の使用を試行する場合、CPU 時間が各コンテナーで要求される CPU の量に基づいて分配されます。

たとえば、あるコンテナーが 500m の CPU 時間を要求し、別のコンテナーが 250m の CPU 時間を要求した場合、ノードで利用可能な追加の CPU 時間は 2:1 の比率でコンテナー間で分配されます。コンテナーが制限を指定している場合、指定した制限を超えて CPU を使用しないようにスロットリングされます。CPU 要求は、Linux カーネルの CFS 共有サポートを使用して適用されます。デフォルトで、CPU 制限は、Linux カーネルの CFS クォータサポートを使用して 100ms の測定間隔で適用されます。ただし、これは無効にすることができます。

7.5.3.1.2. コンテナーのメモリー要求について

コンテナーには要求するメモリー量が保証されます。コンテナーは要求したよりも多くのメモリーを使用できますが、いったん要求した量を超えた場合には、ノードのメモリーが不足している状態では強制終了される可能性があります。コンテナーが要求した量よりも少ないメモリーを使用する場合、システムタスクやデーモンがノードのリソース予約で確保されている分よりも多くのメモリーを必要としない限りそれが強制終了されることはありません。コンテナーがメモリーの制限を指定する場合、その制限量を超えると即時に強制終了されます。

7.5.3.2. オーバーコミットメントと QoS (Quality of Service) クラスについて

ノードは、要求を指定しない Pod がスケジュールされている場合やノードのすべての Pod での制限の合計が利用可能なマシンの容量を超える場合に **オーバーコミット** されます。

オーバーコミットされる環境では、ノード上の Pod がいずれかの時点で利用可能なコンピュートリソースよりも多くの量の使用を試行することができます。これが生じると、ノードはそれぞれの Pod に優先順位を指定する必要があります。この決定を行うために使用される機能は、QoS (Quality of Service) クラスと呼ばれます。

各コンピュートリソースについて、コンテナーは 3 つの QoS クラスに分類されます (優先順位は降順)。

表7.19 QoS (Quality of Service) クラス

優先順位	クラス名	説明
1 (最高)	**Guaranteed**	制限およびオプションの要求がすべてのリソースについて設定されている場合 (0 と等しくない) でそれらの値が等しい場合、コンテナーは **Guaranteed** として分類されます。
2	**Burstable**	制限およびオプションの要求がすべてのリソースについて設定されている場合 (0 と等しくない) でそれらの値が等しくない場合、コンテナーは **Burstable** として分類されます。

優先順位	クラス名	説明
3 (最低)	**BestEffort**	要求および制限がリソースのいずれについても設定されない場合、コンテナーは **BestEffort** として分類されます。

メモリーは圧縮できないリソースであるため、メモリー不足の状態では、最も優先順位の低いコンテナーが最初に強制終了されます。

- **Guaranteed** コンテナーは優先順位が最も高いコンテナーとして見なされ、保証されます。 強制終了されるのは、これらのコンテナーで制限を超えるか、またはシステムがメモリー不足の状態にあるものの、エビクトできる優先順位の低いコンテナーが他にない場合のみです。
- システム不足の状態にある **Burstable** コンテナーは、制限を超過し、**BestEffort** コンテナーが他に存在しない場合に強制終了される可能性があります。
- **BestEffort** コンテナーは優先順位の最も低いコンテナーとして処理されます。これらのコンテナーのプロセスは、システムがメモリー不足になると最初に強制終了されます。

7.5.3.2.1. Quality of Service (QoS) 層でのメモリーの予約方法について

qos-reserved パラメーターを使用して、特定の QoS レベルの Pod で予約されるメモリーのパーセンテージを指定することができます。この機能は、最も低い OoS クラスの Pod が高い QoS クラスの Pod で要求されるリソースを使用できないようにするために要求されたリソースの予約を試行します。

OpenShift Container Platform は、以下のように **qos-reserved** パラメーターを使用します。

- **qos-reserved=memory=100%** の値は、**Burstable** および **BestEffort** QOS クラスが、これらより高い QoS クラスで要求されたメモリーを消費するのを防ぎます。これにより、**Guaranteed** および **Burstable** ワークロードのメモリーリソースの保証レベルを上げることが優先され、**BestEffort** および **Burstable** ワークロードでの OOM が発生するリスクが高まります。
- **qos-reserved=memory=50%** の値は、**Burstable** および **BestEffort** QOS クラスがこれらより高い QoS クラスによって要求されるメモリーの半分を消費することを許可します。
- **qos-reserved=memory=0%** の値は、**Burstable** および **BestEffort** QoS クラスがノードの割り当て可能分を完全に消費することを許可しますが (利用可能な場合)、これにより、**Guaranteed** ワークロードが要求したメモリーにアクセスできなくなるリスクが高まります。この状況により、この機能は無効にされています。

7.5.3.3. swap メモリーと QOS について

QoS (Quality of Service) 保証を維持するため、swap はノード上でデフォルトで無効にすることができます。そうしない場合、ノードの物理リソースがオーバーサブスクライブし、Pod の配置時の Kubernetes スケジューラーによるリソース保証が影響を受ける可能性があります。

たとえば、2 つの Guaranteed pod がメモリー制限に達した場合、それぞれのコンテナーが swap メモリーを使用し始める可能性があります。十分な swap 領域がない場合には、pod のプロセスはシステムのオーバーサブスクライブのために終了する可能性があります。

swap を無効にしないと、ノードが **MemoryPressure** にあることを認識しなくなり、Pod がスケジューリング要求に対応するメモリーを受け取れなくなります。結果として、追加の Pod がノードに配置され、メモリー不足の状態が加速し、最終的にはシステムの Out Of Memory (OOM) イベントが発生するリスクが高まります。

重要

swap が有効にされている場合、利用可能なメモリーについてのリソース不足の処理 (out of resource handling) のエビクションしきい値は予期どおりに機能しなくなります。メモリー不足の状態の場合に Pod をノードからエビクトし、Pod を不足状態にない別のノードで再スケジューリングできるようにリソース不足の処理 (out of resource handling) を利用できるようにします。

7.5.3.4. ノードのオーバーコミットについて

オーバーコミット環境では、最適なシステム動作を提供できるようにノードを適切に設定する必要があります。

ノードが起動すると、メモリー管理用のカーネルの調整可能なフラグが適切に設定されます。カーネルは、物理メモリーが不足しない限り、メモリーの割り当てに失敗することはありません。

この動作を確認するため、OpenShift Container Platform は、**vm.overcommit_memory** パラメーターを **1** に設定し、デフォルトのオペレーティングシステムの設定を上書きすることで、常にメモリーをオーバーコミットするようにカーネルを設定します。

また、OpenShift Container Platform は **vm.panic_on_oom** パラメーターを **0** に設定することで、メモリーが不足したときでもカーネルがパニックにならないようにします。0 の設定は、Out of Memory (OOM) 状態のときに oom_killer を呼び出すようカーネルに指示します。これにより、優先順位に基づいてプロセスを強制終了します。

現在の設定は、ノードに以下のコマンドを実行して表示できます。

```
$ sysctl -a |grep commit
```

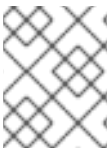
出力例

```
vm.overcommit_memory = 1
```

```
$ sysctl -a |grep panic
```

出力例

```
vm.panic_on_oom = 0
```



注記

上記のフラグはノード上にすでに設定されているはずであるため、追加のアクションは不要です。

各ノードに対して以下の設定を実行することもできます。

- CPU CFS クォータを使用した CPU 制限の無効化または実行
- システムプロセスのリソース予約
- Quality of Service (QoS) 層でのメモリー予約

7.5.3.5. CPU CFS クォータの使用による CPU 制限の無効化または実行

デフォルトで、ノードは Linux カーネルの Completely Fair Scheduler (CFS) クォータのサポートを使用して、指定された CPU 制限を実行します。

CPU 制限の適用を無効にする場合、それがノードに与える影響を理解しておくことが重要になります。

- コンテナーに CPU 要求がある場合、これは Linux カーネルの CFS 共有によって引き続き適用されます。
- コンテナーに CPU 要求がなく、CPU 制限がある場合は、CPU 要求はデフォルトで指定される CPU 制限に設定され、Linux カーネルの CFS 共有によって適用されます。
- コンテナーに CPU 要求と制限の両方がある場合、CPU 要求は Linux カーネルの CFS 共有によって適用され、CPU 制限はノードに影響を与えません。

前提条件

1. 設定するノードタイプの静的な **MachineConfigPool** CRD に関連付けられたラベルを取得します。以下のいずれかの手順を実行します。

   a. マシン設定プールを表示します。

   ```
   $ oc describe machineconfigpool <name>
   ```

   以下に例を示します。

   ```
   $ oc describe machineconfigpool worker
   ```

   出力例

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: MachineConfigPool
   metadata:
     creationTimestamp: 2019-02-08T14:52:39Z
     generation: 1
     labels:
       custom-kubelet: small-pods 1
   ```

   1 ラベルが追加されると、**labels** の下に表示されます。

   b. ラベルが存在しない場合は、キー/値のペアを追加します。

   ```
   $ oc label machineconfigpool worker custom-kubelet=small-pods
   ```

手順

1. 設定変更のためのカスタムリソース (CR) を作成します。

   CPU 制限を無効化する設定例

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: KubeletConfig
   metadata:
   ```

```
  name: disable-cpu-units (1)
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: small-pods (2)
  kubeletConfig:
    cpuCfsQuota: (3)
      - "false"
```

(1) CR に名前を割り当てます。

(2) 設定の変更を適用するラベルを指定します。

(3) **cpuCfsQuota** パラメーターを **false** に設定します。

7.5.3.6. システムリソースのリソース予約

より信頼できるスケジューリングを実現し、ノードリソースのオーバーコミットメントを最小化するために、各ノードでは、クラスターが機能できるようノードで実行する必要のあるシステムデーモン用にそのリソースの一部を予約することができます。とくに、メモリーなどの圧縮できないリソースのリソースを予約することが推奨されます。

手順

Pod 以外のプロセスのリソースを明示的に予約するには、スケジューリングで利用可能なリソースを指定することにより、ノードリソースを割り当てます。詳細については、ノードのリソースの割り当てを参照してください。

7.5.3.7. ノードのオーバーコミットの無効化

有効にされているオーバーコミットを、各ノードで無効にできます。

手順

ノード内のオーバーコミットを無効にするには、そのノード上で以下のコマンドを実行します。

```
$ sysctl -w vm.overcommit_memory=0
```

7.5.4. プロジェクトレベルの制限

オーバーコミットを制御するには、プロジェクトごとのリソース制限の範囲を設定し、オーバーコミットが超過できないプロジェクトのメモリーおよび CPU 制限およびデフォルト値を指定できます。

プロジェクトレベルのリソース制限の詳細は、関連情報を参照してください。

または、特定のプロジェクトのオーバーコミットを無効にすることもできます。

7.5.4.1. プロジェクトでのオーバーコミットメントの無効化

有効にされているオーバーコミットメントをプロジェクトごとに無効にすることができます。たとえば、インフラストラクチャーコンポーネントはオーバーコミットメントから独立して設定できます。

手順

プロジェクト内のオーバーコミットメントを無効にするには、以下の手順を実行します。

1. プロジェクトのオブジェクトファイルを編集します。
2. 以下のアノテーションを追加します。

```
quota.openshift.io/cluster-resource-override-enabled: "false"
```

3. プロジェクトのオブジェクトを作成します。

```
$ oc create -f <file-name>.yaml
```

7.5.5. 関連情報

- プロジェクトごとのリソース制限の設定に関する詳細は、デプロイメントリソースの設定 を参照してください。
- 非 Pod プロセス用にリソースを明示的に予約する方法の詳細については、ノードへのリソースの割り当て を参照してください。

7.6. FEATUREGATE の使用による OPENSHIFT CONTAINER PLATFORM 機能の有効化

管理者は、機能ゲートを使用してデフォルトの機能セットの一部ではない機能を有効にできます。

7.6.1. 機能ゲートについて

FeatureGate カスタムリソース (CR) を使用して、クラスター内の特定の機能セットを有効にすることができます。機能セットは、デフォルトで有効にされない OpenShift Container Platform 機能のコレクションです。

FeatureGate CR を使用して、以下の機能セットをアクティブにすることができます。

- **IPv6DualStackNoUpgrade**.この機能ゲートは、クラスターでデュアルスタックネットワークモードを有効にします。デュアルスタックネットワークは、IPv4 および IPv6 の同時使用をサポートします。この機能セットの有効化は**サポートされておらず**、これを実行すると元に戻すことができなくなり、アップグレードができなくなります。この機能セットは、実稼働クラスターでは推奨されません。

7.6.2. Web コンソールで機能セットの有効化

FeatureGate カスタムリソース (CR) を編集して、OpenShift Container Platform Web コンソールを使用してクラスター内のすべてのノードの機能セットを有効にすることができます。

手順

機能セットを有効にするには、以下を実行します。

1. OpenShift Container Platform Web コンソールで、**Administration → Custom Resource Definitions** ページに切り替えます。
2. **Custom Resource Definitions**ページで、**FeatureGate** をクリックします。
3. **Custom Resource Definition Details**ページで、 **Instances** タブをクリックします。

4. **cluster** 機能ゲートをクリックしてから、**YAML** タブをクリックします。
5. **cluster** インスタンスを編集して特定の機能セットを追加します。

機能ゲートカスタムリソースのサンプル

```
apiVersion: config.openshift.io/v1
kind: FeatureGate
metadata:
  name: cluster (1)
....

spec:
  featureSet: IPv6DualStackNoUpgrade (2)
```

(1) **FeatureGate** CR の名前は **cluster** である必要があります。

(2) **IPv6DualStackNoUpgrade** 機能セットを追加して、デュアルスタックネットワークモードを有効にします。

変更を保存すると、新規マシン設定が作成され、マシン設定プールが更新され、変更が適用されている間に各ノードのスケジューリングが無効になります。

注記

IPv6DualStackNoUpgrade 機能セットを有効にすると、元に戻すことができず、更新もできなくなります。この機能セットは、実稼働クラスターでは推奨されません。

検証

ノードが Ready 状態に戻ると、ノードの **kubelet.conf** ファイルを確認して機能ゲートが有効になっていることを確認できます。

1. Web コンソールの **Administrator** パースペクティブで、**Compute → Nodes** に移動します。
2. ノードを選択します。
3. **Node details** ページで **Terminal** をクリックします。
4. ターミナルウィンドウで、root ディレクトリーをホストに切り替えます。

```
sh-4.2# chroot /host
```

5. **kubelet.conf** ファイルを表示します。

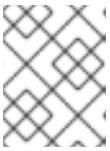
```
sh-4.2# cat /etc/kubernetes/kubelet.conf
```

出力例

```
 ...
featureGates:
  InsightsOperatorPullingSCA: true,
```

```
  LegacyNodeRoleBehavior: false
...
```

true として一覧表示されている機能は、クラスターで有効になっています。



注記

一覧表示される機能は、OpenShift Container Platform のバージョンによって異なります。

7.6.3. CLI を使用した機能セットの有効化

FeatureGate カスタムリソース (CR) を編集し、OpenShift CLI (**oc**) を使用してクラスター内のすべてのノードの機能セットを有効にすることができます。

前提条件

- OpenShift CLI (**oc**) がインストールされている。

手順

機能セットを有効にするには、以下を実行します。

1. **cluster** という名前の **FeatureGate** CR を編集します。

```
$ oc edit featuregate cluster
```

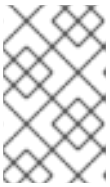
FeatureGate カスタムリソースのサンプル

```
apiVersion: config.openshift.io/v1
kind: FeatureGate
metadata:
  name: cluster (1)
spec:
  featureSet: IPv6DualStackNoUpgrade (2)
```

(1) **FeatureGate** CR の名前は **cluster** である必要があります。

(2) **IPv6DualStackNoUpgrade** 機能セットを追加して、デュアルスタックネットワークモードを有効にします。

変更を保存すると、新規マシン設定が作成され、マシン設定プールが更新され、変更が適用されている間に各ノードのスケジューリングが無効になります。



注記

IPv6DualStackNoUpgrade 機能セットを有効にすると、元に戻すことができず、更新もできなくなります。この機能セットは、実稼働クラスターでは推奨されません。

検証

ノードが Ready 状態に戻ると、ノードの **kubelet.conf** ファイルを確認して機能ゲートが有効になっていることを確認できます。

1. ノードのデバッグセッションを開始します。

```
$ oc debug node/<node_name>
```

2. ルートディレクトリーをホストに切り替えます。

```
sh-4.2# chroot /host
```

3. **kubelet.conf** ファイルを表示します。

```
sh-4.2# cat /etc/kubernetes/kubelet.conf
```

出力例

```
 ...
featureGates:
  InsightsOperatorPullingSCA: true,
  LegacyNodeRoleBehavior: false
 ...
```

true として一覧表示されている機能は、クラスターで有効になっています。

注記

一覧表示される機能は、OpenShift Container Platform のバージョンによって異なります。

第8章 ネットワークエッジ上にあるリモートワーカーノード

8.1. ネットワークエッジでのリモートワーカーノードの使用

ネットワークエッジにあるノードで OpenShift Container Platform クラスターを設定できます。このトピックでは、**リモートワーカーノード** と呼ばれます。リモートワーカーノードを含む通常のクラスターは、オンプレミスのマスターとワーカーノードを、クラスターに接続する他の場所にあるワーカーノードと統合します。このトピックは、リモートワーカーノードの使用のベストプラクティスに関するガイダンスを提供することを目的としており、特定の設定に関する詳細情報は含まれません。

リモートワーカーノードでのデプロイメントパターンの使用に関しては、さまざまな業界 (通信、小売、製造、政府など) で複数のユースケースがあります。たとえば、リモートワーカーノードを Kubernetes ゾーン に統合して、プロジェクトとワークロードを切り離し、分離することができます。

ただし、リモートワーカーノードを使用すると、高いレイテンシーの発生や、ネットワーク接続が断続的に失われるなどの問題が発生する可能性があります。リモートワーカーノードを含むクラスターの課題には、以下のようなものがあります。

- **ネットワーク分離**: OpenShift Container Platform コントロールプレーンとリモートワーカーノードは、相互に通信できる必要があります。コントロールプレーンとリモートワーカーノードの間に距離があるため、ネットワークの問題が発生すると、この通信が妨げられる可能性があります。OpenShift Container Platform が、ネットワークの分離に対応する方法や、クラスターへの影響を軽減する方法についての詳細は、リモートワーカーノードのネットワークの分離 について参照してください。

- **停電**: コントロールプレーンとリモートワーカーノードは別々の場所にあるため、リモートの場所での停電、またはそれぞれの場所からの任意の場所での停電により、クラスターに悪影響を及ぼす可能性があります。OpenShift Container Platform がノードの電源の損失に対応する方法や、クラスターへの影響を軽減する方法についての詳細は、リモートワーカーノードでの電源の損失 について参照してください。

- **急激な高レイテンシーまたは一時的なスループットの低下**: ネットワークの場合と同様に、クラスターとリモートワーカーノード間のネットワーク状態の変更は、クラスターに悪影響を及ぼす可能性があります。このような状況は、本書の対象外となります。

リモートワーカーノードを含むクラスターを計画する場合には、以下の制限に注意してください。

- リモートワーカーノードは、ユーザーによってプロビジョニングされるインフラストラクチャーのベアメタルクラスターでのみサポートされます。

- OpenShift Container Platform は、オンプレミスクラスターが使用するクラウドプロバイダー以外のクラウドプロバイダーを使用するリモートワーカーノードをサポートしません。

- ワークロードを 1 つの Kubernetes ゾーンから別の Kubernetes ゾーンに移動すると、(特定のタイプのメモリーが異なるゾーンで利用できないなどの) システムや環境に関する課題により、問題が発生する可能性があります。

- プロキシーおよびファイアウォールでは、本書では扱われていない追加的制限が出てくる可能性があります。このような制限への対処方法については、ファイアウォールの設定 など、関連する OpenShift Container Platform のドキュメントを参照してください。

- コントロールプレーンとネットワークエッジノード間の L2/L3 レベルのネットワーク接続を設定および維持する必要があります。

8.1.1. リモートワーカーノードによるネットワーク分離

すべてのノードは、10 秒ごとに OpenShift Container Platform クラスターの Kubernetes Controller Manager Operator (kube コントローラー) にハートビートを送信します。クラスターがノードからハートビートを受信しない場合、OpenShift Container Platform は複数のデフォルトメカニズムを使用して応答します。

OpenShift Container Platform は、ネットワークパーティションやその他の中断に対して回復性を持たせるように設計されています。ソフトウェアのアップグレードの中断、ネットワーク分割、ルーティングの問題など、より一般的な中断の一部を軽減することができます。軽減策には、リモートワーカーノードの Pod が正しい CPU およびメモリーリソースの量を要求すること、適切なレプリケーションポリシーの設定、ゾーン間の冗長性の使用、ワークロードでの Pod の Disruption Budget の使用などが含まれます。

設定した期間後に kube コントローラーのノードとの接続が解除された場合、コントロールプレーンのノードコントローラーはノードの正常性を **Unhealthy** に更新し、ノードの **Ready** 状態を **Unknown** とマークします。この操作に応じて、スケジューラーはそのノードへの Pod のスケジューリングを停止します。オンプレミスノードコントローラーは、effect が **NoExecute** の **node.kubernetes.io/unreachable** テイントをノードに追加し、デフォルトで 5 分後に、エビクション用にノード上で Pod をスケジュールします。

Deployment オブジェクト、または **StatefulSet** オブジェクトなどのワークロードコントローラーが、正常でないノードの Pod にトラフィックを転送し、他のノードがクラスターに到達できる場合、OpenShift Container Platform はトラフィックをノードの Pod から遠ざけます。クラスターに到達できないノードは、新しいトラフィックルーティングでは更新されません。その結果、それらのノードのワークロードは、正常でないノードに到達しようとします。

以下の方法で接続損失の影響を軽減できます。

- デーモンセットを使用したテイントを容認する Pod の作成
- ノードがダウンした場合に自動的に再起動する静的 Pod の使用
- Kubernetes ゾーンを使用した Pod エビクションの制御
- Pod のエビクションを遅延または回避するための Pod 容認の設定
- ノードを正常でないとマークするタイミングを制御するように kubelet を設定します。

リモートワーカーノードを含むクラスターでこれらのオブジェクトを使用する方法の詳細は、About remote worker node strategies を参照してください。

8.1.2. リモートワーカーノードの電源損失

リモートワーカーノードの電源がなくなったり、強制的な再起動を行う場合、OpenShift Container Platform は複数のデフォルトメカニズムを使用して応答します。

設定した期間後に Kubernetes Controller Manager Operator (kube コントローラー) のノードとの接続が解除された場合、コントロールプレーンはノードの正常性を **Unhealthy** に更新し、ノードの **Ready** 状態を **Unknown** とマークします。この操作に応じて、スケジューラーはそのノードへの Pod のスケジューリングを停止します。オンプレミスノードコントローラーは、effect が **NoExecute** の **node.kubernetes.io/unreachable** テイントをノードに追加し、デフォルトで 5 分後に、エビクション用にノード上で Pod をスケジュールします。

ノードでは、ノードが電源を回復し、コントロールプレーンに再接続する際に、Pod を再起動する必要があります。

注記

再起動時に Pod をすぐに再起動する必要がある場合は、静的 Pod を使用します。

ノードの再起動後に kubelet も再起動し、ノードにスケジュールされた Pod の再起動を試行します。コントロールプレーンへの接続にデフォルトの 5 分よりも長い時間がかかる場合、コントロールプレーンはノードの正常性を更新して **node.kubernetes.io/unreachable** テイントを削除することができません。ノードで、kubelet は実行中の Pod をすべて終了します。これらの条件がクリアされると、スケジューラーはそのノードへの Pod のスケジューリングを開始できます。

以下の方法で、電源損失の影響を軽減できます。

- デーモンセットを使用したテイントを容認する Pod の作成
- ノードを使用して自動的に再起動する静的 Pod の使用
- Pod のエビクションを遅延または回避するための Pod 容認の設定
- ノードコントローラーがノードを正常でないとマークするタイミングを制御するための kubelet の設定

リモートワーカーノードを含むクラスターでこれらのオブジェクトを使用する方法の詳細は、About remote worker node strategies を参照してください。

8.1.3. リモートワーカーノードストラテジー

リモートワーカーノードを使用する場合は、アプリケーションを実行するために使用するオブジェクトを考慮してください。

ネットワークの問題や電源の損失時に必要とされる動作に基づいて、デーモンセットまたは静的 Pod を使用することが推奨されます。さらに、Kubernetes ゾーンおよび容認を使用して、コントロールプレーンがリモートワーカーノードに到達できない場合に Pod エビクションを制御したり、回避したりできます。

デーモンセット

デーモンセットは、以下の理由により、リモートワーカーノードでの Pod の管理に最適な方法です。

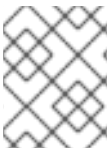
- デーモンセットは通常、動作の再スケジュールを必要としません。ノードがクラスターから切断される場合、ノードの Pod は実行を継続できます。OpenShift Container Platform はデーモンセット Pod の状態を変更せず、Pod を最後に報告された状態のままにします。たとえば、デーモンセット Pod が **Running** 状態の際にノードが通信を停止する場合、Pod は実行し続けますが、これは OpenShift Container Platform によって実行されていることが想定されます。
- デーモンセット Pod はデフォルトで、**tolerationSeconds** 値のない **node.kubernetes.io/unreachable** テイントおよび **node.kubernetes.io/not-ready** テイントの **NoExecute** 容認で作成されます。これらのデフォルト値により、コントロールプレーンがノードに到達できなくても、デーモンセット Pod がエビクトされることはありません。以下に例を示します。

  デフォルトでデーモンセット Pod に容認を追加

```
tolerations:
  - key: node.kubernetes.io/not-ready
    operator: Exists
    effect: NoExecute
```

```
- key: node.kubernetes.io/unreachable
  operator: Exists
  effect: NoExecute
- key: node.kubernetes.io/disk-pressure
  operator: Exists
  effect: NoSchedule
- key: node.kubernetes.io/memory-pressure
  operator: Exists
  effect: NoSchedule
- key: node.kubernetes.io/pid-pressure
  operator: Exists
  effect: NoSchedule
- key: node.kubernetes.io/unschedulable
  operator: Exists
  effect: NoSchedule
```

- デーモンセットは、ワークロードが一致するワーカーノードで実行されるように、ラベルを使用することができます。
- OpenShift Container Platform サービスエンドポイントを使用してデーモンセット Pod の負荷を分散できます。



注記

デーモンセットは、OpenShift Container Platform がノードに到達できない場合、ノードの再起動後に Pod をスケジュールしません。

静的 Pod

ノードの再起動時に Pod を再起動する必要がある場合 (電源が切れた場合など)、静的な Pod を考慮してください。ノードの kubelet は、ノードの再起動時に静的 Pod を自動的に再起動します。

注記

静的 Pod はシークレットおよび設定マップを使用できません。

Kubernetes ゾーン

Kubernetes ゾーン は、速度を落としたり、または場合によっては Pod エビクションを完全に停止したりすることができます。

コントロールプレーンがノードに到達できない場合、デフォルトでノードコントローラーは **node.kubernetes.io/unreachable** テイントを適用し、1秒あたり 0.1 ノードのレートで Pod をエビクトします。ただし、Kubernetes ゾーンを使用するクラスターでは、Pod エビクションの動作が変更されます。

ゾーンのすべてのノードに **False** または **Unknown** の **Ready** 状態が見られる、ゾーンが完全に中断された状態の場合、コントロールプレーンは **node.kubernetes.io/unreachable** テイントをそのゾーンのノードに適用しません。

(ノードの 55% 超が **False** または **Unknown** 状態である) 部分的に中断されたゾーンの場合、Pod のエビクションレートは 1 秒あたり 0.01 ノードに低減されます。50 未満の小規模なクラスターにあるノードにテイントは付けられません。これらの動作を有効にするには、クラスターに 4 つ以上のゾーンが必要です。

ノード仕様に **topology.kubernetes.io/region** ラベルを適用して、ノードを特定のゾーンに割り当てます。

Kubernetes ゾーンのノードラベルの例

```
kind: Node
apiVersion: v1
metadata:
  labels:
    topology.kubernetes.io/region=east
```

KubeletConfig オブジェクト

kubelet が各ノードの状態をチェックする時間を調整することができます。

オンプレミスノードコントローラーがノードを **Unhealthy** または **Unreachable** 状態にマークするタイミングに影響を与える間隔を設定するには、**node-status-update-frequency** および **node-status-report-frequency** パラメーターが含まれる **KubeletConfig** オブジェクトを作成します。

各ノードの kubelet は **node-status-update-frequency** 設定で定義されたノードのステータスを判別し、**node-status-report-frequency** 設定に基づいてそのステータスをクラスターに報告します。デフォルトで、kubelet は 10 秒ごとに Pod のステータスを判別し、毎分ごとにステータスを報告します。ただし、ノードの状態が変更されると、kubelet は変更をクラスターに即時に報告します。OpenShift Container Platform は、ノードリース機能ゲートが有効にされている場合にのみ **node-status-report-frequency** 設定を使用します。これは OpenShift Container Platform クラスターのデフォルト状態です。ノードリース機能ゲートが無効にされている場合、ノードは **node-status-update-frequency** 設定に基づいてそのステータスを報告します。

kubelet 設定の例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: disable-cpu-units
spec:
  machineConfigPoolSelector:
    matchLabels:
      machineconfiguration.openshift.io/role: worker (1)
  kubeletConfig:
    node-status-update-frequency: (2)
      - "10s"
    node-status-report-frequency: (3)
      - "1m"
```

(1) **MachineConfig** オブジェクトのラベルを使用して、この **KubeletConfig** オブジェクトが適用されるノードタイプを指定します。

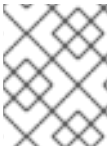
(2) kubelet がこの **MachineConfig** オブジェクトに関連付けられたノードのステータスをチェックする頻度を指定します。デフォルト値は **10s** です。このデフォルト値を変更すると、**node-status-report-frequency** の値は同じ値に変更されます。

(3) kubelet がこの **MachineConfig** オブジェクトに関連付けられたノードのステータスを報告する頻度を指定します。デフォルト値は **1m** です。

node-status-update-frequency パラメーターは **node-monitor-grace-period** および **pod-eviction-timeout** パラメーターと共に機能します。

- **node-monitor-grace-period** パラメーターは、コントローラーマネージャーがハートビートを受信しない場合に、この **MachineConfig** オブジェクトに関連付けられたノードが **Unhealthy** とマークされた後に、OpenShift Container Platform が待機する時間を指定します。この待機時間後も、ノード上のワークロードは引き続き実行されます。**node-monitor-grace-period** の期限が切れた後にリモートワーカーノードがクラスターに再度加わる場合、Pod は実行を継続します。新規 Pod をノードにスケジュールできます。**node-monitor-grace-period** の間隔は **40s** です。**node-status-update-frequency** の値は、**node-monitor-grace-period** の値よりも低い値である必要があります。

- **pod-eviction-timeout** パラメーターは、**MachineConfig** オブジェクトに関連付けられたノードを **Unreachable** としてマークした後、エビクション用に Pod のマークを開始するまでに OpenShift Container Platform が待機する時間を指定します。エビクトされた Pod は、他のノードで再スケジュールされます。**pod-eviction-timeout** の期限が切れた後にリモートワーカーノードがクラスターに再結合する場合、ノードコントローラーがオンプレミスで Pod をエビクトしたため、リモートワーカーノードで実行されている Pod は終了します。続いて、Pod をそのノードに再スケジュールできます。**pod-eviction-timeout** の期間は **5m0s** です。



注記

node-monitor-grace-period および **pod-eviction-timeout** パラメーターを変更することはサポートされていません。

容認

オンプレミスノードコントローラーが、effect が **NoExecute** の **node.kubernetes.io/unreachable** テイントを到達できないノードに追加する場合、Pod 容認を使用して effect を軽減することができます。

effect が **NoExecute** のテイントは、ノードですでに実行中の Pod に以下のような影響を及ぼします。

- テイントを容認しない Pod は、エビクションのキューに置かれます。
- 容認の仕様に **tolerationSeconds** 値を指定せずにテイントを容認する Pod は、永久にバインドされたままになります。
- 指定された **tolerationSeconds** 値でテイントを容認する Pod は、指定された期間バインドされます。時間が経過すると、Pod はエビクションのキューに置かれます。

effect が **NoExecute** の **node.kubernetes.io/unreachable** テイントおよび **node.kubernetes.io/not-ready** テイントで Pod の容認を設定し、Pod のエビクションを遅延したり回避したりできます。

Pod 仕様での容認の例

```
...
tolerations:
- key: "node.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute" 1
- key: "node.kubernetes.io/not-ready"
  operator: "Exists"
  effect: "NoExecute" 2
  tolerationSeconds: 600
...
```

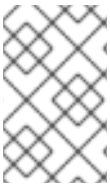
1 **tolerationSeconds** のない **NoExecute** effect により、コントロールプレーンがノードに到達する場合は Pod が永続的に残ります。

2 **tolerationSeconds**: 600 の **NoExecute** effect により、コントロールプレーンがノードに **Unhealthy** のマークを付ける場合に Pod が 10 分間そのまま残ります。

OpenShift Container Platform は、**pod-eviction-timeout** 値の経過後、**tolerationSeconds** 値を使用します。

OpenShift Container Platform オブジェクトの他のタイプ

レプリカセット、デプロイメント、およびレプリケーションコントローラーを使用できます。スケジューラーは、ノードが 5 分間切断された後、これらの Pod を他のノードに再スケジュールできます。他のノードへの再スケジュールは、管理者が特定数の Pod を確実に実行し、アクセスできるようにする REST API などの一部のワークロードにとって有益です。



注記

リモートワーカーノードを使用する際に、リモートワーカーノードが特定の機能用に予約されることが意図されている場合、異なるノードでの Pod の再スケジュールは許容されない可能性があります。

ステートフルセット は、停止時に再起動されません。Pod は、コントロールプレーンが Pod の終了を認識できるまで、**terminating** 状態のままになります。

同じタイプの永続ストレージにアクセスできないノードにスケジュールしないようにするため、OpenShift Container Platform では、ネットワークの分離時に永続ボリュームを必要とする Pod を他のゾーンに移行することはできません。

関連情報

- DaemonSet に関する詳細は、DaemonSets を参照してください。
- テイントおよび容認についての詳細は、ノードテイントを使用した Pod 配置の制御 を参照してください。
- **KubeletConfig** オブジェクトの設定に関する詳細は、KubeletConfig CRD の作成 を参照してください。
- レプリカセットについての詳細は、 ReplicaSets を参照してください。
- デプロイメントについての詳細は、デプロイメント を参照してください。
- レプリケーションコントローラーについての詳細は、レプリケーションコントローラー を参照してください。
- コントローラーマネージャーの詳細は、Kubernetes Controller Manager Operator を参照してください。