



OpenShift Container Platform 4.1

ノード

OpenShift Container Platform 4.1 でのノードの設定および管理

OpenShift Container Platform 4.1 ノード

OpenShift Container Platform 4.1 でのノードの設定および管理

法律上の通知

Copyright © 2020 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

本書では、クラスターのノード、Pod、コンテナ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 POD の使用	3
1.1. POD の使用	3
1.2. POD の表示	6
1.3. OPENSIFT CONTAINER PLATFORM クラスターでの POD の設定	7
1.4. POD の自動スケーリング	12
1.5. POD への機密性の高いデータの提供	24
1.6. 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	32
1.7. POD スケジューリングの決定に POD の優先順位を含める	35
1.8. ノードセクターの使用による特定ノードへの POD の配置	40
第2章 POD のノードへの配置の制御 (スケジューリング)	44
2.1. スケジューラーによる POD 配置の制御	44
2.2. デフォルトスケジューラーの設定による POD 配置の制御	45
2.3. アフィニティルールと非アフィニティルールの使用による他の POD との相対での POD の配置	61
2.4. ノードのアフィニティルールを使用したノード上での POD 配置の制御	68
2.5. POD のオーバーコミットノードへの配置	74
2.6. ノードテイントを使用した POD 配置の制御	75
2.7. ノードセクターの使用による特定ノードへの POD の配置	83
第3章 ジョブと DEAMONSET の使用	91
3.1. DEAMONSET 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	91
3.2. ジョブの使用による POD でのタスクの実行	93
第4章 ノードの使用	101
4.1. OPENSIFT CONTAINER PLATFORM クラスター内のノードの閲覧と一覧表示	101
4.2. ノードの使用	105
4.3. ノードの管理	108
4.4. ノードあたりの POD の最大数の管理	110
4.5. NODE TUNING OPERATOR の使用	112
4.6. ノードの再起動について	118
4.7. 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	120
4.8. OPENSIFT CONTAINER PLATFORM クラスター内のノードのリソースの割り当て	124
4.9. ノード監査ログの表示	128
第5章 コンテナの使用	133
5.1. コンテナについて	133
5.2. POD のデプロイ前の、INIT コンテナの使用によるタスクの実行	133
5.3. ボリュームの使用によるコンテナデータの永続化	135
5.4. PROJECTED ボリュームによるボリュームのマッピング	141
5.5. コンテナによる API オブジェクト使用の許可	148
5.6. OPENSIFT CONTAINER PLATFORM コンテナへの/からのファイルのコピー	156
5.7. OPENSIFT CONTAINER PLATFORM コンテナでのリモートコマンドの実行	158
5.8. コンテナ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	160
第6章 クラスターの操作	163
6.1. OPENSIFT CONTAINER PLATFORM クラスター内のシステムイベント情報の表示	163
6.2. OPENSIFT CONTAINER PLATFORM のノードが保持できる POD の数の見積り	172
6.3. コンテナメモリとリスク要件を満たすためのクラスターメモリの設定	176
6.4. 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	182
6.5. 機能ゲートの使用による各種機能の有効化	189

第1章 POD の使用

1.1. 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タ単位です。

1.1.1. Pod について

Pod はコンテナ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 には独自の内部 IP アドレスが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、コンテナが終了するまで実行されるか、その他の理由でコンテナ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なく、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。

1.1.2. Pod 設定の例

OpenShift Container Platform は、Pod の Kubernetes の概念を使用します。Pod はホスト上にデプロイされる1つ以上のコンテナであり、定義され、デプロイされ、管理される最小のコンピュータ単位です。

以下は Pod のサンプル定義です。これは、統合コンテナイメージレジストリーという OpenShift Container Platform インフラストラクチャーの一部で、長期間実行されるサービスを提供します。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: ①
    app: hello-openshift
  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
      terminationMessagePolicy: File
      image: registry.redhat.io/openshift4/ose-ogging-eventrouter:v4.1 ⑥
  serviceAccount: default ⑦
  volumes: ⑧
    - 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。この例で使用されているラベルは **registry=default** です。
- 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、コンテナが特権付きコンテナ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** はコンテナ定義の配列を指定します。この場合 (ほとんどの場合)、これは1つのみになります。
- 5 コンテナは外部ストレージボリュームがコンテナ内にマウントされるかどうかを指定します。この場合、レジストリーのデータを保存するためのボリュームと、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用のボリュームがあります。
- 6 Pod 内の各コンテナは、独自のコンテナイメージからインスタンス化されます。
- 7 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 が認証する必要が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

チャータ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。

- 8 Pod は、コンテナ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、レジストリーストレージの一時的なボリュームおよびサービスアカウントの認証情報が含まれる **secret** ボリュームが提供されます。



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。

1.2. 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。

1.2.1. 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タ単位です。Pod はコンテナ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、

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。

1.2.2. 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. 次のコマンドを実行します。

```
$ oc get pods
```

例:

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

```

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>

```

1.2.3. Pod の使用状況についての統計の表示

コンテナ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. 次のコマンドを実行します。

```
$ oc adm top pods
```

例:

```

$ oc adm top pods -n openshift-console
NAME                CPU(cores)  MEMORY(bytes)
console-7f58c69899-q8c8k  0m          22Mi
console-7f58c69899-xhbgg  0m          25Mi
downloads-594fcccf94-bcwk8  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。

1.3. OPENSIFT CONTAINER PLATFORM クラスタでの POD の設定

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

クラスタ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。

1.3.1. 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または失敗したコンテナ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
(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	Daemonset	すべて

Pod のコンテナが失敗し、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、コンテナが再起動します。コンテナ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「[Example States](#)」を参照してください。

1.3.2. 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に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>
```

1.3.3. 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 (クラスタのスケールダウンまたはクラスタ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構成されています。

- 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
```

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。

1.3.3.1. 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。

- 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
```

- PodDisruptionBudget** は **policy/v1beta1** API グループの一部です。
- 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。
- 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>
```

1.3.4. 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 Critical にするには、以下を実行します。

- Pod 仕様を作成するか、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
```

- 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。

1. Pod を作成します。

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

1.4. 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。

1.4.1. 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。



重要

メモリー使用率の自動スケーリングはテクノロジープレビュー機能のみとして提供されています。

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。クエリーとスケールアップ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までに 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、このスケールアップ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、スケールアップはデプロイメント設定のレプリカ数に直接対応します。自動スケールアップは **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ルアップを防ぎます。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。

1.4.1.1. 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1.1 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。

1.4.2. CPU 使用率のための Horizontal Pod Autoscaler の作成

指定する CPU 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 または ReplicationController オブジェクトについて Horizontal Pod Autoscaler (HPA) を作成できます。

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、実行する必要がある 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
```

```
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-
kubernetes-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. 以下のいずれかの手順を実行します。

- 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 DeploymentConfig の **HorizontalPodAutoscaler** オブジェクトを作成します。

```

$ oc autoscale dc/<dc-name> \ 1
--min <number> \ 2
--max <number> \ 3
--cpu-percent=<percent> 4

```

- 1 DeploymentConfig 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。
- 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。

- 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 ReplicationController の **HorizontalPodAutoscaler** オブジェクトを作成します。

```

$ oc autoscale rc/<rc-name> \ 1
--min <number> \ 2
--max <number> \ 3
--cpu-percent=<percent> 4

```

- 1 ReplicationController 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。

- 2 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。
- 特定の CPU 値についてスケーリングするには、既存の DeploymentConfig または ReplicationController について以下のような 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: v1 3
    kind: ReplicationController 4
    name: example 5
  minReplicas: 1 6
  maxReplicas: 10 7
  metrics: 8
  - type: Resource
    resource:
      name: cpu 9
      target:
        type: Utilization 10
        averageValue: 500Mi 11

```

- 1 Use the **autoscaling/v2beta2** API.
- 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 については、**v1** を使用します。
 - DeploymentConfig 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 CPU 使用率に **cpu** を指定します。
- 10 **Utilization** に設定します。
- 11 タイプを **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 hpa-resource-metrics-memory
```

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

以下のコマンドは、**image-registry** DeploymentConfig が制御する Pod の 3 から 7 までのレプリカを維持し、すべての Pod で 75% の平均 CPU 使用率を維持する Horizontal Pod Autoscaler を作成します。

```
$ oc autoscale dc/image-registry --min 3 --max 7 --cpu-percent=75
deploymentconfig "image-registry" autoscaled
```

このコマンドは、以下の定義で Horizontal Pod Autoscaler を作成します。

```
$ oc edit hpa frontend -n openshift-image-registry
```

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  creationTimestamp: "2020-02-21T20:19:28Z"
  name: image-registry
  namespace: default
  resourceVersion: "32452"
  selfLink: /apis/autoscaling/v1/namespaces/default/horizontalpodautoscalers/frontend
  uid: 1a934a22-925d-431e-813a-d00461ad7521
spec:
  maxReplicas: 7
  minReplicas: 3
  scaleTargetRef:
    apiVersion: apps.openshift.io/v1
    kind: DeploymentConfig
    name: image-registry
  targetCPUUtilizationPercentage: 75
status:
  currentReplicas: 5
  desiredReplicas: 0
```

以下の例は、**image-registry** DeploymentConfig 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 get dc image-registry
NAME          REVISION  DESIRED  CURRENT  TRIGGERED BY
image-registry 1         3        3        config

$ oc autoscale dc/image-registry --min=5 --max=7 --cpu-percent=75
horizontalpodautoscaler.autoscaling/image-registry autoscaled

$ oc get dc image-registry
NAME          REVISION  DESIRED  CURRENT  TRIGGERED BY
image-registry 1         5        5        config
```

1.4.3. メモリー使用率のための 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 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。

重要

メモリー使用率の自動スケーリングはテクノロジープレビュー機能としてのみ提供されています。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。これらの機能は、近々発表予定の製品機能をリリースに先駆けてご提供することにより、お客様は機能性をテストし、開発プロセス中にフィードバックをお寄せいただくことができます。

Red Hat のテクノロジープレビュー機能のサポートについての詳細は、<https://access.redhat.com/support/offerings/techpreview/>を参照してください。

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
```

```
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。

- 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ついてスケーリングするには、既存の DeploymentConfig または ReplicationController について以下のような **HorizontalPodAutoscaler** オブジェクトを作成します。

```

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: Utilization ⑩
        averageValue: 500Mi ⑪

```

- ① Use the **autoscaling/v2beta2** API.
- 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 については、**v1** を使用します。
 - DeploymentConfig 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 メモリ使用率に **metrics** パラメーターを使用します。
 - 9 メモリ使用率の **memory** を指定します。
 - 10 タイプを **Utilization** 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。
- 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。

```

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:
  - type: Resource
    resource:
      name: memory 9
      target:
        type: Utilization 10
        averageValue: 50 11

```

- 1 Use the **autoscaling/v2beta2** API.
- 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 については、**v1** を使用します。

- DeploymentConfig 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 メモリ使用率に **metrics** パラメーターを使用します。
- 9 メモリ使用率の **memory** を指定します。
- 10 **Utilization** に設定します。
- 11 **averageUtilization** または **averageValue** およびメモリー値を指定します。

2. Horizontal Pod Autoscaler を作成します。

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

例:

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

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

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

```

1.4.4. 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
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。

```

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。

```

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: unable to get metrics for resource cpu: no metrics returned from heapster

```

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。

```

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

```

1.4.4.1. 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

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>
```

例:

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

条件は、出力の **Conditions** フィールドに表示されます。

```
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

```

1.4.5. 追加リソース

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、「[Understanding Deployments and DeploymentConfigs](#)」を参照してください。

1.5. 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。

1.5.1. 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 シークレットオブジェクト定義

```

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 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** マップのエントリが base64 に変換され、このエントリは自動的に **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構成されます。

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)。

1.5.1.1. 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はその種類のキー/値の要件に従うことが意図されていることを示します。

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。

1.5.1.2. シークレット設定の例

以下は、シークレットの設定ファイルのサンプルです。

4つのファイルを作成する YAML シークレット

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  username: dmFsdWUtMQ0K 1
  password: dmFsdWUtMQ0KDQo= 2
stringData:
  hostname: myapp.mydomain.com 3
secret.properties: |- 4
  property1=valueA
  property2=valueB
```

- 1** デコードされる値が含まれるファイル
- 2** デコードされる値が含まれるファイル
- 3** 提供される文字列が含まれるファイル
- 4** 提供されるデータが含まれるファイル

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:
      # name must match the volume name below
      - name: secret-volume
        mountPath: /etc/secret-volume
        readOnly: true
  volumes:
  - name: secret-volume
    secret:
      secretName: test-secret
  restartPolicy: Never
```

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

シークレットデータと環境変数が投入されたビルド設定の YAML

```
apiVersion: v1
kind: BuildConfig
metadata:
  name: secret-example-bc
spec:
  strategy:
    sourceStrategy:
      env:
      - name: TEST_SECRET_USERNAME_ENV_VAR
        valueFrom:
          secretKeyRef:
            name: test-secret
            key: username
```

1.5.1.3. 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。

1.5.2. 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。

- 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)。

1.5.2.1. 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** タイプのオブジェクトを実際に参照していることを確認する必要があります。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。

1.5.2.2. 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。このシークレットでは、任意の値を含む、構造化されていない **key:value** ペアを利用できます。

手順

1. マスターの YAML ファイルにシークレットオブジェクトを作成します。

例:

```
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シークレットオブジェクトを作成します。

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

次に、以下を実行します。

1. このシークレットを使ってこのシークレットの参照を許可したい、Pod のサービスアカウントを更新します。
2. 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)。

1.5.3. 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 を使用することがあります)。

シークレットの更新は、新規コンテナ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。 **kubectrl 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。今後はコントローラーが古い **resourceVersion** を使用して再起動できるよう 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。

1.5.4. 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の Pod 仕様

```
apiVersion: v1
kind: Service
metadata:
  name: registry
  annotations:
    service.alpha.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。

1.5.4.1. 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.alpha.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.alpha.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.alpha.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。

```
$ oc get secrets

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

$ oc describe secret my-service-pod
Name:         my-service-pod
Namespace:    openshift-console
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: builder
              kubernetes.io/service-account.uid: ab-11e9-988a-0eb4e1b4a396

Type: kubernetes.io/service-account-token

Data
ca.crt: 5802 bytes
namespace: 17 bytes
token:
```



```
$ oc delete secret <secret_name>
$ oc annotate service <service_name> service.alpha.openshift.io/serving-cert-generation-error-
$ oc annotate service <service_name> service.alpha.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の後に - を付けます。

1.6. 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。

1.6.1. 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 resetting 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、[vendor/k8s.io/kubernetes/pkg/kubelet/cm/deviceplugin/device_plugin_stub.go](https://github.com/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。

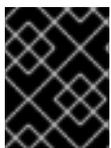
1.6.1.1. デバイスプラグインのデプロイ方法

- DaemonSet 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。

1.6.2. 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の **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。

1.6.3. 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. 設定したいノードタイプの静的な Machine Config Pool 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 を表示します。

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

例:

```
# oc describe machineconfig 00-worker

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

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、Custom Resource) 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

kube
letconfig.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。

1.7. 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ブションを有効にできます。Pod の優先順位は、他の Pod との対比で Pod の重要度を示し、その優先順位に基づいて Pod をキューに入れます。Pod のプリエンブションは、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ブ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 をスケジューリングできます。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ブ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。

プリエンブションはスケジューラー設定ファイルの **disablePreemption** パラメーターで制御されます。これはデフォルトで **false** に設定されます。

1.7.1. Pod の優先順位について

Pod の優先順位およびプリエンブ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リングされる可能性があります。Pod をスケジューリングできない場合、スケジューラーは引き続き他の優先順位の低い Pod をスケジューリングします。

1.7.1.1. 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ブションは、1000000000 (10 億) 以下の 32 ビットの整数値を取ることができます。プリエンブ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
NAME                CREATED AT
cluster-logging     2019-03-13T14:45:12Z
system-cluster-critical 2019-03-13T14:01:10Z
system-node-critical 2019-03-13T14:01:10Z
```

- **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。



注記

既存クラスターをアップグレードする場合、既存 Pod の優先順位はゼロになります。ただし、**scheduler.alpha.kubernetes.io/critical-pod** アノテーションを持つ既存 Pod は **system-cluster-critical** クラスに自動的に変換されます。このアノテーションを持つ Fluentd クラスターのロギング Pod は、**cluster-logging** 優先順位クラスに変換されません。

1.7.1.2. 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 を作成できません。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。

1.7.2. Pod のプリエンブションについて

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ブ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、プリエンブ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 のプリエンブ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ブ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 のプリエンブ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 のプリエンブションを実行し、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。

プリエンブションは、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 のプリエンブションを実行するノードを考慮します。

1.7.2.1. Pod プリエンブ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ブ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 のプリエンブ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 のプリエンブ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 のプリエンブ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 のプリエンブ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。

1.7.2.2. 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。

1.7.3. 優先順位およびプリエンプションの設定

priorityClassName を Pod 仕様で使用し、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 の優先順位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/v1beta1
kind: PriorityClass
metadata:
  name: high-priority ①
value: 1000000 ②
globalDefault: false ③
description: "This priority class should be used for XYZ service pods only." ④
```

- 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。
- ④ 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 ❶

```

❶ 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。

1.7.4. 優先順位およびプリエンプシヨンの無効化

Pod の優先順位およびプリエンプシヨン機能を無効にできます。

この機能が無効にされた後に、既存 Pod はそれらの優先順位フィールドを維持しますが、プリエンプシヨンは無効にされ、優先順位フィールドは無視されます。この機能が無効にされると、新規 Pod に優先順位クラス名を設定できません。



重要

クラスターがリソース不足の状態にある場合、Critical Pod のスケジューリングにはスケジューラーのプリエンプシヨンが使用されます。このため、プリエンプシヨンを無効にしないことが推奨されています。DaemonSet Pod は DaemonSet コントローラーによってスケジュールされるため、プリエンプシヨンを無効にしても影響を受けません。

手順

クラスターのプリエンプシヨンを無効にするには、以下を実行します。

1. スケジューラー Operator カスタムリソースを編集して **disablePreemption: true** パラメーターを追加します。

```
oc edit scheduler cluster
```

```

apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  creationTimestamp: '2019-03-12T01:45:02Z'
  generation: 1
  name: example

```

```
resourceVersion: '1882034'
selfLink: /apis/config.openshift.io/v1/schedulers/example
uid: 743701e9-4468-11e9-bd34-02a7fe1bf828
spec:
  disablePreemption: true
```

1.8. ノードセクターの使用による特定ノードへの POD の配置

ノードセクターは、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とノードセ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。

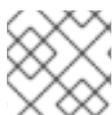
1.8.1. ノードセクターの使用による Pod 配置の制御

Pod にノードセクターのラベルを使用することで、Pod がスケジュールされる場所を制御できます。

ノードセクターにより、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。

ラベルはノードまたは MachineConfig に追加できますが、ノードまたはマシンが停止するとラベルは維持されません。MachineSet にラベルを追加すると、新しいノードまたはマシンにそのラベルが追加されます。

ノードセクターを既存 Pod に追加するには、ノードセクターを ReplicaSet、Daemonset、または StatefulSet などのノード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、ノードセクターを Pod 仕様に直接追加できます。



注記

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

前提条件

ノードセクターを既存 Pod に追加する必要がある場合、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** ReplicaSet 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. 必要なラベルをノードに追加します。

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

ラベルはノードに適用されます。

```
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:
    beta.kubernetes.io/os: linux
    failure-domain.beta.kubernetes.io/zone: us-east-1a
    node.openshift.io/os_version: '4.1'
    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
    region: east ①
    beta.kubernetes.io/arch: amd64
    type: user-node ②
  ...
```

- ① ② ノードに追加するラベルを指定します。

または、ラベルを MachineSet に追加できます。

```
$ oc edit MachineSet abc612-msrtw-worker-us-east-1c
```

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

```

....

spec:
  replicas: 2
  selector:
    matchLabels:
      machine.openshift.io/cluster-api-cluster: ci-ln-89dz2y2-d5d6b-4995x
      machine.openshift.io/cluster-api-machine-role: worker
      machine.openshift.io/cluster-api-machine-type: worker
      machine.openshift.io/cluster-api-machineset: ci-ln-89dz2y2-d5d6b-4995x-worker-us-
east-1a
  template:
    metadata:
      creationTimestamp: null
    labels:
      machine.openshift.io/cluster-api-cluster: ci-ln-89dz2y2-d5d6b-4995x
      machine.openshift.io/cluster-api-machine-role: worker
      machine.openshift.io/cluster-api-machine-type: worker
      machine.openshift.io/cluster-api-machineset: ci-ln-89dz2y2-d5d6b-4995x-worker-us-
east-1a
    spec:
      metadata:
        creationTimestamp: null
      labels:
        region: east ①
        type: user-node ②
....

```

① ②

ノードに追加するラベルを指定します。

2. 必要なノードセクターを Pod を追加します。

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

例:

```

kind: ReplicaSet

....

spec:

....

template:
  metadata:
    creationTimestamp: null
  labels:
    ingresscontroller.operator.openshift.io/deployment-ingresscontroller: default
    pod-template-hash: 66d5cf9464

```

```
spec:
  nodeSelector:
    beta.kubernetes.io/os: linux
    node-role.kubernetes.io/worker: "
  type: user-node ①
```

① 必要なノードセレクターを追加します。

- 新規 Pod の場合、セレクターを Pod 仕様に直接追加できます。

```
apiVersion: v1
kind: Pod
...
spec:
  nodeSelector:
    <key>: <value>
...

```

例:

```
apiVersion: v1
kind: Pod
....
spec:
  nodeSelector:
    region: east
    type: user-node
```

注記

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。

第2章 POD のノードへの配置の制御 (スケジューリング)

2.1. 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。

2.1.1. 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。

2.1.1.1. 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。

2.1.1.2. 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リング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 はスケジューリングされません。

Pod がスケジューリング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の使用および Pod のアフィニティおよび非アフィニティの使用について参照してください。

これらの高度なスケジューリング機能を使うと、管理者は Pod をスケジューリング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。

2.1.1.3. 非アフィニティ

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を設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。特定レベルの非アフィニティ(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 を配置しようとしています。

Pod がスケジューリング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の使用および Pod のアフィニティおよび非アフィニティの使用について参照してください。

これらの高度なスケジューリング機能を使うと、管理者は Pod をスケジューリング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。

2.2. 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ています。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。

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: ❶
    name: scheduler-policy
  defaultNodeSelector: type=user-node,region=east ❷
```

❶ 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 オプションで、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。

2.2.1. 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せられます。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。

2.2.1.1. 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 ConfigMap のサンプル

```
apiVersion: v1
data:
  policy.cfg: |
    {
      "kind": "Policy",
      "apiVersion": "v1",
      "predicates": [
        {"name": "MaxGCEPDVolumeCount"},
        {"name": "GeneralPredicates"},
        {"name": "MaxAzureDiskVolumeCount"},
        {"name": "MaxCSIVolumeCountPred"}
      ]
    }

```

```

    {"name": "CheckVolumeBinding"},
    {"name": "MaxEBSVolumeCount"},
    {"name": "PodFitsResources"},
    {"name": "MatchInterPodAffinity"},
    {"name": "CheckNodeUnschedulable"},
    {"name": "NoDiskConflict"},
    {"name": "NoVolumeZoneConflict"},
    {"name": "MatchNodeSelector"},
    {"name": "HostName"},
    {"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`

```

2.2.2. 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。その後、JSON ファイルから ConfigMap を生成し、ConfigMap 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": [ ❶
    {"name": "PodFitsHostPorts"},
    {"name": "PodFitsResources"},
    {"name": "NoDiskConflict"},
    {"name": "NoVolumeZoneConflict"},
    {"name": "MatchNodeSelector"},

```

```

{"name": "MaxEBSVolumeCount"},
{"name": "MaxAzureDiskVolumeCount"},
{"name": "checkServiceAffinity"},
{"name": "PodToleratesNodeNoExecuteTaints"},
{"name": "MaxGCEPDVolumeCount"},
{"name": "MatchInterPodAffinity"},
{"name": "PodToleratesNodeTaints"},
{"name": "HostName"}
],
"priorities": [ 2
{"name": "LeastRequestedPriority", "weight": 1},
{"name": "BalancedResourceAllocation", "weight": 1},
{"name": "ServiceSpreadingPriority", "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 ConfigMap を作成します。

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

- 1 ConfigMap の名前を入力します。

例:

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

```

apiVersion: v1
data:
  policy.cfg: |
    {
      "kind": "Policy",
      "apiVersion": "v1",
      "predicates": [
        {"name": "MaxGCEPDVolumeCount"},
        {"name": "GeneralPredicates"},
        {"name": "MaxAzureDiskVolumeCount"},
        {"name": "MaxCSIVolumeCountPred"},
        {"name": "CheckVolumeBinding"},
        {"name": "MaxEBSVolumeCount"},
        {"name": "PodFitsResources"},
        {"name": "MatchInterPodAffinity"},
        {"name": "CheckNodeUnschedulable"},
        {"name": "NoDiskConflict"},
        {"name": "NoVolumeZoneConflict"},
        {"name": "MatchNodeSelector"},
        {"name": "HostName"},

```

```

    {"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`

```

3. スケジューラー Operator カスタムリソースを編集して ConfigMap を追加します。

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

- ❶ ConfigMap の名前を指定します。

例:

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

スケジューラー設定リソースに変更を加えた後に、**opensift-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
```

例:

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

```

Creating scheduler with fit predicates 'map[MaxGCEPDVolumeCount:{}
MaxAzureDiskVolumeCount:{} CheckNodeUnschedulable:{} NoDiskConflict:{}
NoVolumeZoneConflict:{} MatchNodeSelector:{} GeneralPredicates:{}
MaxCSIVolumeCountPred:{} CheckVolumeBinding:{} MaxEBSVolumeCount:{}

```

```
PodFitsResources:{} MatchInterPodAffinity:{} HostName:{} PodToleratesNodeTaints:{}}' and
priority functions 'map[InterPodAffinityPriority:{} LeastRequestedPriority:{}
ServiceSpreadingPriority:{} ImageLocalityPriority:{} SelectorSpreadPriority:{} EqualPriority:{}
BalancedResourceAllocation:{} NodePreferAvoidPodsPriority:{} NodeAffinityPriority:{}
TaintTolerationPriority:{}]
```

2.2.3. 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 ConfigMap 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。スケジューラーポリシーを作成するには、述語と優先順位の追加および削除を ConfigMap 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 ConfigMap を編集します。

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

例:

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

```
apiVersion: v1
data:
  policy.cfg: |
    {
      "kind": "Policy",
      "apiVersion": "v1",
      "predicates": [ ❶
        {"name": "MaxGCEPDVolumeCount"},
        {"name": "GeneralPredicates"},
        {"name": "MaxAzureDiskVolumeCount"},
        {"name": "MaxCSIVolumeCountPred"},
        {"name": "CheckVolumeBinding"},
        {"name": "MaxEBSVolumeCount"},
        {"name": "PodFitsResources"},
        {"name": "MatchInterPodAffinity"},
        {"name": "CheckNodeUnschedulable"},
        {"name": "NoDiskConflict"},
        {"name": "NoVolumeZoneConflict"},
        {"name": "MatchNodeSelector"},
        {"name": "HostName"},
        {"name": "PodToleratesNodeTaints"}
      ],
      "priorities": [ ❷
        {"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 ConfigMap を削除します。

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

例:

```
$ 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。

例:

```
$ vi policy.cfg
```

```

apiVersion: v1
data:
  policy.cfg: |
    {
      "kind" : "Policy",
      "apiVersion" : "v1",
      "predicates" : [
        {"name" : "PodFitsHostPorts"},
        {"name" : "PodFitsResources"},
        {"name" : "NoDiskConflict"},
        {"name" : "NoVolumeZoneConflict"},
        {"name" : "MatchNodeSelector"},
        {"name" : "MaxEBSVolumeCount"},
        {"name" : "MaxAzureDiskVolumeCount"},
        {"name" : "CheckVolumeBinding"},
        {"name" : "CheckServiceAffinity"},
        {"name" : "PodToleratesNodeNoExecuteTaints"},
        {"name" : "MaxGCEPDVolumeCount"},
        {"name" : "MatchInterPodAffinity"},
        {"name" : "PodToleratesNodeTaints"},

```

```

    {"name" : "HostName"}
  ],
  "priorities" : [
    {"name" : "LeastRequestedPriority", "weight" : 2},
    {"name" : "BalancedResourceAllocation", "weight" : 2},
    {"name" : "ServiceSpreadingPriority", "weight" : 2},
    {"name" : "EqualPriority", "weight" : 2}
  ]
}

```

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 ConfigMap を再作成します。

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

- ❶ ConfigMap の名前を入力します。

例:

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

2.2.3.1. 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。

2.2.3.1.1. 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。

2.2.3.1.1.1. 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 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 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 は Azure ディスクボリュームの最大数を確認します。

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

PodToleratesNodeTaints は Pod がノードテイントを許容できるかどうかを確認します。

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

CheckNodeUnschedulable 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 は、バインドされている PVC とバインドされていない PVC の両方の場合に Pod が要求するボリュームに基づいて適しているかどうかを評価します。* バインドされている PVC については、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によって満たされていることを確認します。* バインドされていない PVC については、述語は PVC 要件を満たす PV を検索し、PV のノードアフィニティーが指定ノードによって満たされていることを確認します。

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"}
```

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

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

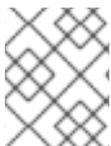
MaxCSIVolumeCountPred

MatchInterPodAffinity 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"}
```

2.2.3.1.1.2. 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 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 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 は、ServiceAffinity ラベルがノードでスケジューラされる Pod について同種のものであることを確認します。

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

PodToleratesNodeNoExecuteTaints 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"}
```

2.2.3.1.2. 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 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 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 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 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"}
```

2.2.3.2. 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。

2.2.3.2.1. 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 Configmap に指定されます。

2.2.3.2.1.1. 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 は、ノードアフィニティーのスケジューリング設定に応じてノードの優先順位を決定します。

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

TaintTolerationPriority 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 は、Pod コンテナ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)、およびステータスフルなセットを検索し、次にそれらのセ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 は、ノードの対応する PodAffinityTerm が満たされている場合に **weightedPodAffinityTerm** 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 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 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 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}
```

2.2.3.2.1.2. 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 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 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、ノードスケジューラされる 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 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}
```

2.2.3.2.2. 設定可能な優先順位

これらの優先順位を **openshift-config** プロジェクトのスケジューラーポリシー Configmap に設定し、優先順位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 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>" ❸
    }
}
]
}

```

- ❶ 優先度の名前を指定します。
- ❷ 重みを指定します。ゼロ以外の正の値を指定します。
- ❸ 一致するラベルを指定します。

例:

```

{
  "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** 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。

```

{
  "kind": "Policy",
  "apiVersion": "v1",
  "priorities": [
    {
      "name": "<name>", ❶
      "weight": 1 ❷
      "argument": {
        "labelPreference": {
          "label": "<label>", ❸
          "presence": true ❹
        }
      }
    }
  ]
}

```

```

    }
  ]
}

```

- 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。

2.2.4. 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": {
          "label": "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つのト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": {
          "label": "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": {
        "label": "region"
      }
    },
    {
      "name": "RequireRegion",
      "argument": {
        "labelsPresence": {
          "label": "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}
  ]
}

```

2.3. 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。

2.3.1. 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ます、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 設定のサンプル

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

```

metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity: ❶
      requiredDuringSchedulingIgnoredDuringExecution: ❷
      - labelSelector:
          matchExpressions:
            - key: security ❸
              operator: In ❹
              values:
                - S1 ❺
          topologyKey: failure-domain.beta.kubernetes.io/zone
  containers:
    - name: with-pod-affinity
      image: docker.io/ocpqe/hello-pod

```

- ❶ Pod のアフィニティーを設定するためのスタンザです。
- ❷ required (必須) ルールを定義します。
- ❸ ❺ 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。
- ❹ 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 設定のサンプル

```

apiVersion: v1
kind: Pod
metadata:
  name: with-pod-antiaffinity
spec:
  affinity:
    podAntiAffinity: ❶
      preferredDuringSchedulingIgnoredDuringExecution: ❷
      - weight: 100 ❸
        podAffinityTerm:
          labelSelector:
            matchExpressions:
              - key: security ❹
                operator: In ❺
                values:
                  - S2
            topologyKey: kubernetes.io/hostname
  containers:
    - name: with-pod-affinity
      image: docker.io/ocpqe/hello-pod

```

- ❶ Pod の非アフィニティーを設定するためのスタンザです。
- ❷ 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。

2.3.2. 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. **podAntiAffinity** スタンザを使用し、**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
```

2.3.3. 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
```

2.3.4. 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 のアフィニティおよび非アフィニティについて示しています。

2.3.4.1. Pod のアフィニティ

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。

2.3.4.2. 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** が付けられていません。

```
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。

2.3.4.3. 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。

```
NAME    READY   STATUS    RESTARTS   AGE    IP      NODE
pod-s2  0/1     Pending  0           32s    <none>
```

2.4. 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 Platform node では、アフィニティ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クターを使って定義されます。

2.4.1. ノードのアフィニティについて

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

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。

2.4.2. 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。

```
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
```

2.4.3. 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リング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
```

3. **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** を使用してラベルをノードで必要になるようにします。
4. Pod を作成します。

```
$ oc create -f e2e-az3.yaml
```

2.4.4. ノードのアフィニティルールの例

以下の例は、ノードのアフィニティを示しています。

2.4.4.1. 一致するラベルを持つノードのアフィニティ

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 **pod-s1** 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
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 **pod-s1** を **Node1** にスケジュールできます。

```
$ oc get pod -o wide
NAME    READY   STATUS    RESTARTS  AGE   IP      NODE
pod-s1  1/1     Running   0          4m   IP1    node1
```

2.4.4.2. 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 **pod-s1** 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
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 `pod-s1` は `Node1` にスケジュールすることができません。

```
$ oc describe pod pod-s1
<---snip--->
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).
```

2.4.5. 追加リソース

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。

2.5. 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。

2.5.1. 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の制限と要求を調整することができます。



注記

コンテナ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 `LimitRanges` を注意して設定してください。

2.5.2. 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とはありません。

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
vm.overcommit_memory = 1
```

```
$ sysctl -a |grep panic
vm.panic_on_oom = 0
```



注記

上記のフラグはノードに常に設定されているので、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

2.6. 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。

2.6.1. 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。

テイントはノード仕様 (**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。ノードのテイントはノードに対し、テイントを容認しないすべての Pod を拒否するよう指示します。

テイントおよび容認は、key、value、および effect で構成されています。演算子により、これらの3つのパラメーターのいずれかを空のままにすることができます。

表2.1 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。 <table border="1" data-bbox="518 891 1428 1646"> <tbody> <tr> <td>NoSchedule</td> <td> <ul style="list-style-type: none"> 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。 </td> </tr> <tr> <td>PreferNoSchedule</td> <td> <ul style="list-style-type: none"> 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。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。 </td> </tr> <tr> <td>NoExecute</td> <td> <ul style="list-style-type: none"> 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。 </td> </tr> </tbody> </table>	NoSchedule	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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。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	<ul style="list-style-type: none"> 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。
NoSchedule	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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。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	<ul style="list-style-type: none"> 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。 						
operator	<table border="1" data-bbox="518 1736 1428 1975"> <tbody> <tr> <td>Equal</td> <td>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。</td> </tr> <tr> <td>Exists</td> <td>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。</td> </tr> </tbody> </table>	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。		
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。						

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 kubernetes に組み込まれています。

- **node.kubernetes.io/not-ready**: ノードは準備状態ではありません。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/out-of-disk**: 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。これはノード条件 **OutOfDisk=True** 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。

2.6.1.1. 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 仕様に **tolerationSeconds** パラメーターを指定して、Pod がエビクト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 **NoExecute** のあるテイントがノードに追加される場合、テイントを容認しない Pod は即時にエビクトされます (テイントを容認する Pod はエビクトされません)。ただし、エビクトされる Pod に **tolerationSeconds** パラメーターがある場合、Pod は期間切れになるまでエビクトされません。

例:

```
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、一致するテイントがない場合、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。

2.6.1.2. 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。

- Pod に一致する容認のあるテイントを処理します。
- 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。
 - 容認の仕様に **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。

例:

- ノードには以下のテイントがあります。

```
$ 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。

```
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。

2.6.1.3. ノードの問題の発生時における Pod エビクションの禁止

OpenShift Container Platform は、**node unreachable** および **node not ready** 状態をテイントとして表示するよう設定できます。これにより、デフォルトの 5 分を使用するのではなく、unreachable (到達不能) または not ready (準備ができていない) 状態になるノードにバインドされたままになる期間を Pod 仕様ごとに指定することができます。

テイントベースのエビクション機能はデフォルトで有効にされています。テイントはノードコントローラーによって自動的に追加され、Pod を **Ready** ノードからエビクトするための通常のロジックは無効にされます。

- ノードが not ready (準備ができていない) 状態になると、**node.kubernetes.io/not-ready:NoExecute** テイントは追加され、Pod はノードでスケジュールできなくなります。既存 Pod は容認期間 (秒数) 中はそのまま残ります。
- ノードが not reachable (到達不能) の状態になると、**node.kubernetes.io/unreachable:NoExecute** テイントは追加され、Pod はノードでスケジュールできません。既存 Pod は容認期間 (秒数) 中はそのまま残ります。

この機能により、**tolerationSeconds** と組み合わせることで、Pod は問題のいずれかまたは両方を持つノードにどの程度の期間バインドされるか指定することができます。

2.6.1.4. 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

OpenShift Container Platform は、メモリー不足やディスク不足のような状態を報告した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。この **Taint Nodes By Condition** 機能は、デフォルトで有効にされます。

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。

DaemonSet 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/out-of-disk` (Critical Pod の場合のみ)
- `node.kubernetes.io/unschedulable` (1.10 以降)
- `node.kubernetes.io/network-unavailable` (ホストネットワークのみ)

DaemonSet には任意の容認を追加することも可能です。

2.6.1.5. 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、**not-ready** や **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、容認する Pod はエビクトされません。



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。

この機能は、**tolerationSeconds** と組み合わせることで、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、Pod はレートが制限された方法でエビクトされます。**tolerationSeconds** の期間前にこの状態が解消される場合、Pod は削除されません。

OpenShift Container Platform は、**node.kubernetes.io/not-ready** および **node.kubernetes.io/unreachable** の容認を、Pod の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
  tolerations:
  - key: node.kubernetes.io/not-ready
    operator: Exists
    effect: NoExecute
    tolerationSeconds: 300
  - key: node.kubernetes.io/unreachable
    operator: Exists
    effect: NoExecute
    tolerationSeconds: 300
```

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。

DaemonSet Pod は、tolerationSeconds のない以下のテイントの NoExecute 容認で作成されます。

- **node.kubernetes.io/unreachable**
- **node.kubernetes.io/not-ready**

これにより、DaemonSet Pod は、**DefaultTolerationSeconds** 受付コントローラーが無効化されていても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。

2.6.2. テイントおよび容認 (Toleration) の追加

テイントをノードに、容認 (Toleration) を Pod 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。

手順

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パラメーターを使って、以下のコマンドを使用します。

```
$ oc adm taint nodes <node-name> <key>=<value>:<effect>
```

例:

```
$ oc adm taint nodes node1 key1=value1:NoExecute
```

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。

- Pod 仕様を **tolerations** セクション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

```
tolerations:
- key: "key1" ①
  operator: "Equal" ②
  value: "value1" ③
  effect: "NoExecute" ④
  tolerationSeconds: 3600 ⑤
```

① ② ③ ④ 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。

⑤ **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。

例:

Exists 演算子を含む Pod 設定ファイルのサンプル

```
tolerations:
- key: "key1"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

これらの容認のいずれも上記の **oc adm taint** コマンドで作成される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。

2.6.2.1. テイントおよび容認 (Toleration) 使ってノードをユーザー専用にする

ノードのセットを特定のユーザーセットが排他的に使用するよう指定できます。

手順

専用ノードを指定するには、以下を実行します。

- テイントをそれらのノードに追加します。

例:

```
$ 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対応する容認を Pod に追加します。
容認のある Pod のみが専用ノードを使用することを許可されます。

2.6.2.2. 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

特定ユーザーが専用ノードのみを使用できるようにノードを設定することができます。

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. テイントをそれらのノードに追加します。
例:

```
$ 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対応する容認を Pod に追加します。
受付コントローラーは、Pod が **key:value** ラベル (**dedicated=groupName**) が付けられたノードのみにスケジューラされるようにノードのアフィニティを追加します。
3. テイントと同様のラベル (**key:value** ラベルなど) を専用ノードに追加します。

2.6.2.3. 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(GPU など) 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。

手順

Pod が特殊ハードウェアからブロックされるようにするには、以下を実行します。

1. 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
$ oc adm taint nodes <node-name> disktype=ssd:PreferNoSchedule
```

2. 受付コントローラーを使用して、特別なハードウェアを使用する Pod に対応する容認を追加します。

たとえば受付コントローラーは容認を追加することで、Pod の一部の機能を使用し、Pod が特殊ノードを使用できるかどうかを判別できます。

Pod が特殊ハードウェアのみを使用できるようにするには、追加のメカニズムが必要です。たとえば、特殊ハードウェアを持つノードにラベルを付け、ハードウェアを必要とする Pod でノードのアフィニティを使用できます。

2.6.3. 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。

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>-
```

例:

```
$ oc adm taint nodes ip-10-0-132-248.ec2.internal key1-
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。

```
tolerations:
- key: "key2"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

2.7. ノードセクターの使用による特定ノードへの POD の配置

ノードセクターは、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 で指定されたセクターを使って定義されます。ノードセクターを使用して特定の Pod を特定のノードに配置したり、すべての Pod を特定ノードのプロジェクトに配置したりするか、またはデフォルトノードセクターを作成して、デフォルトのノードセクターまたはプロジェクトセクターのない Pod をスケジュールすることができます。

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とノードセ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。

2.7.1. ノードセクターの使用による Pod 配置の制御

Pod にノードセクターのラベルを使用することで、Pod がスケジュールされる場所を制御できます。

ノードセクターにより、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。

ラベルはノードまたは MachineConfig に追加できますが、ノードまたはマシンが停止するとラベルは維持されません。MachineSet にラベルを追加すると、新しいノードまたはマシンにそのラベルが追加されます。

ノードセクターを既存 Pod に追加するには、ノードセクターを ReplicaSet、Daemonset、または StatefulSet などのノード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、ノードセクターを Pod 仕様に直接追加できます。



注記

ノードセクターを既存のスケジュールされている Pod に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 に追加する必要がある場合、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** ReplicaSet 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. 必要なラベルをノードに追加します。

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

ラベルはノードに適用されます。

```
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:
    beta.kubernetes.io/os: linux
    failure-domain.beta.kubernetes.io/zone: us-east-1a
    node.openshift.io/os_version: '4.1'
    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
    region: east ❶
```

```
beta.kubernetes.io/arch: amd64
type: user-node ②
```

```
....
```

① ② ノードに追加するラベルを指定します。

または、ラベルを MachineSet に追加できます。

```
$ oc edit MachineSet abc612-msrtw-worker-us-east-1c
```

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

....

spec:
  replicas: 2
  selector:
    matchLabels:
      machine.openshift.io/cluster-api-cluster: ci-ln-89dz2y2-d5d6b-4995x
      machine.openshift.io/cluster-api-machine-role: worker
      machine.openshift.io/cluster-api-machine-type: worker
      machine.openshift.io/cluster-api-machineset: ci-ln-89dz2y2-d5d6b-4995x-worker-us-
east-1a
  template:
    metadata:
      creationTimestamp: null
      labels:
        machine.openshift.io/cluster-api-cluster: ci-ln-89dz2y2-d5d6b-4995x
        machine.openshift.io/cluster-api-machine-role: worker
        machine.openshift.io/cluster-api-machine-type: worker
        machine.openshift.io/cluster-api-machineset: ci-ln-89dz2y2-d5d6b-4995x-worker-us-
east-1a
    spec:
      metadata:
        creationTimestamp: null
      labels:
        region: east ①
        type: user-node ②
....
```

① ②

ノードに追加するラベルを指定します。

2. 必要なノードセクターを Pod を追加します。

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

例:

```
kind: ReplicaSet
....
spec:
....

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

❶ 必要なノードセレクターを追加します。

- 新規 Pod の場合、セレクターを Pod 仕様に直接追加できます。

```
apiVersion: v1
kind: Pod
...

spec:
  nodeSelector:
    <key>: <value>
...

```

例:

```
apiVersion: v1
kind: Pod
....

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



注記

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。

2.7.2. クラスタ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。

クラスター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 は該当する **<key>:<value>** を追加し、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。



注記

Pod の **<key>:<value>** ペアを追加できます。ただし、デフォルトの **<key>** に異なる **<value>** を追加することはできません。

たとえば、クラスターノードセレクターが **region: east** の場合、以下の Pod 仕様は新規ペアを追加し、これが許可されます。

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

以下の Pod 仕様は **region** に異なる値を使用しており、これは許可されません。

```
spec:
  nodeSelector:
    region: west
```

Pod を作成してい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ノードセレクターよりも優先されます。

手順

デフォルトのクラスターノード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 カスタムリソースを編集し、クラスターノードセレクターを追加します。

```
$ oc edit scheduler cluster
```

```

apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  name: cluster
spec: {}
policy:
spec:
  defaultNodeSelector: type=user-node,region=east ❶

```

- ❶ 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. ノードまたは MachineSet を編集し、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 ip-10-0-142-25.ec2.internal type=user-node region=east
```

MachineSet にラベルを付けるには、以下を実行します。

```
$ oc label MachineSet abc612-msrtw-worker-us-east-1c type=user-node region=east
```

Pod を作成する際に、OpenShift Container Platform は適切な **<key>:<value>** を追加し、Pod をラベルが付けられたノードでスケジューリングします。

例:

```

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

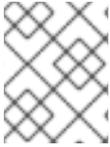
```

2.7.3. プロジェクトスコープのノードセレクターの作成

namespace 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。

プロジェクトノードセレクターを使用する場合、namespace で Pod を作成すると、OpenShift Container Platform は該当する **<key>:<value>** を追加し、一致するラベルのあるノードで Pod をスケジューリングします。

ラベルはノードまたは MachineConfig に追加できますが、ノードまたはマシンが停止するとラベルは維持されません。MachineSet にラベルを追加すると、新しいノードまたはマシンにそのラベルが追加されます。



注記

Pod の **<key>:<value>** ペアを追加できます。ただし、デフォルトの **<key>** に異なる **<value>** を追加することはできません。

たとえば、プロジェクトノードセクターが **region: east** の場合、以下の Pod 仕様は新規ペアを追加し、これが許可されます。

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

以下の Pod 仕様は **region** に異なる値を使用しており、これは許可されません。

```
spec:
  nodeSelector:
    region: west
```

クラスタースコープのデフォルトノードセクターがない場合、プロジェクトノードセクターが優先されます。

手順

デフォルトのプロジェクトノードセ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、またはプロジェクトに関連付けられた既存の namespace を編集し、**openshift.io/node-selector** パラメーターを追加します。

```
$ oc edit namespace <name>
```

```
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    openshift.io/node-selector: "type=user-node,region=east" ❶
    openshift.io/sa.scc.mcs: s0:c17,c14
    openshift.io/sa.scc.supplemental-groups: 1000300000/10000
    openshift.io/sa.scc.uid-range: 1000300000/10000
  creationTimestamp: 2019-06-10T14:39:45Z
  labels:
    openshift.io/run-level: "0"
  name: demo
  resourceVersion: "401885"
  selfLink: /api/v1/namespaces/openshift-kube-apiserver
  uid: 96ecc54b-8b8d-11e9-9f54-0a9ae641edd0
spec:
  finalizers:
    - kubernetes
status:
  phase: Active
```

- ❶ 適切な **<key>:<value>** ペアを持つ `openshift.io/node-selector`` を追加します。

2. ノードまたは MachineSet を編集し、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 ip-10-0-142-25.ec2.internal type=user-node region=east
```

MachineSet にラベルを付けるには、以下を実行します。

```
$ oc label MachineSet abc612-msrtw-worker-us-east-1c type=user-node region=east
```

Pod を namespace に作成する際に、OpenShift Container Platform は適切な **<key>:<value>** を追加し、Pod をラベルが付けられたノードでスケジューリングします。

例:

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

第3章 ジョブと DEAMONSET の使用

3.1. DEAMONSET 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

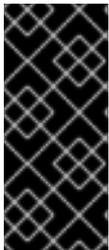
管理者は、DeamonSet を作成して OpenShift Container Platform クラスター内の特定の、またはすべてのノードで Pod のレプリカを実行するために使用できます。

DeamonSet 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。DeamonSet を削除すると、DeamonSet によって作成された Pod がクリーンアップされます。

DeamonSet 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。

セキュリティ上の理由から、クラスター管理者のみが DeamonSet を作成できます。

daemonset 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

Daemonset 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、daemonset 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。

3.1.1. デフォルトスケジューラーによるスケジューリング

DeamonSet 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 DeamonSet Pod は DeamonSet コントローラーが作成し、スケジューリングしていました。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、DeamonSet 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、DeamonSet 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 は、OpenShift Container Platform ではデフォルトで有効にされています。これにより、`.spec.nodeName` の term ではなく `NodeAffinity` の term を DeamonSet Pod に追加することで、DeamonSet 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 DeamonSet をスケジューリング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。DeamonSet 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。DeamonSet コントローラーは、DeamonSet Pod を作成または修正する場合にのみこれらの操作を実行し、DeamonSet 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 DaemonSet 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、DaemonSet Pod をスケジューリングするときに、スケジューリングできないノードを無視します。

3.1.2. DaemonSet の作成

daemonset の作成時に、**nodeSelector** フィールドは daemonset 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

- DaemonSet 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 のプロジェクト全体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、**oc adm new-project <name> --node-selector=""** を使用してデフォルトのノードセレクターを上書きします。

手順

DaemonSet を作成するには、以下を実行します。

1. daemonset yaml ファイルを定義します。

```
apiVersion: extensions/v1beta1
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 daemonset 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クターです。上記のラベルセ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. daemonset 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
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
Node: openshift-node01.hostname.com/10.14.20.134
$ oc describe pod/hello-daemonset-e3md9|grep Node
Node: openshift-node02.hostname.com/10.14.20.137
```

重要

- DeamonSet 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。
- DeamonSet を削除してから、異なるテンプレートと同じラベルセクターを使用して新規の DeamonSet 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、DeamonSet 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。

daemonset 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。

3.2. 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 のレプリカがリニアップされます。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。

3.2.1. 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

Cron 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、ジョブの実行方法をユーザーに指定させることで、定期的なジョブにビルドされます。Cron 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。

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。



警告

Cron ジョブはスケジュールの実行時間ごとに約1回ずつジョブ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可能性のある場合があります。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。

3.2.2. 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構成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。
 - ワークキューを使用する並列ジョブでは、未設定のままにしておきます。未設定の場合、デフォルトは **parallelism** 値に設定されます。

3.2.2.1. 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。このフィールドは秒単位で指定し、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、最長期間は有効ではありません。

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。

3.2.2.2. 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。

3.2.2.3. 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 つのフィールドは 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.failedJobsHistoryLimit**。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

- 必要がなくなった 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。

3.2.3. 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。

3.2.4. 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
```

- 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、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。
- 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、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。
- 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。
- 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。
- 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。
- 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「[Example States](#)」を参照してください。

- ジョブを作成します。

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

注記

oc run 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 run pi --image=perl --replicas=1 --restart=OnFailure \
  --command -- perl -Mbignum=bpi -wle 'print bpi(2000)'
```

3.2.5. 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 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 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。

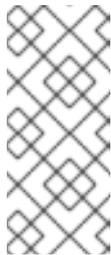
③ ③ ③ ジョブを開始するためのオプションの期限 (秒単位)(何らかの理由によりスケジュールされた時間が経過する場合)。ジョブの実行が行われない場合、ジョブの失敗としてカウントされます。これが指定されない場合は期間が設定されません。

④ ④ 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 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 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 run` 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 run pi --image=perl --schedule='*/1 * * * *' \  
  --restart=OnFailure --labels parent="cronjobpi" \  
  --command -- perl -Mbignum=bpi -wle 'print bpi(2000)'
```

`oc run` 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。

Cron ジョブの作成時に、`oc run` は **Never** または **OnFailure** 再起動ポリシー (`--restart`) のみをサポートします。

第4章 ノードの使用

4.1. OPENSIFT 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。

4.1.1. 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
```

```
NAME                STATUS  ROLES  AGE   VERSION
master.example.com  Ready   master  7h    v1.13.4+b626c2fe1
node1.example.com   Ready   worker  7h    v1.13.4+b626c2fe1
node2.example.com   Ready   worker  7h    v1.13.4+b626c2fe1
```

- **-wide** オプションは、すべてのノードに関する追加情報を提供します。

```
$ oc get nodes -o wide
```

- 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>
```

これらのコマンドの出力にある **STATUS** 列には、ノードの以下の状態が表示されます。

表4.1 ノードの状態

条件	説明
Ready	ノードは、 True を返すことでその準備状況を apiserver に報告します。
NotReady	コンテナ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 をスケジューリングすることができません。

- 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>
```

例:

```

$ oc describe node node1.example.com

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://1.13.9-1.rhaos4.1.gitd70609a.el8
Kubelet Version: v1.13.4+b626c2fe1
Kube-Proxy Version: v1.13.4+b626c2fe1
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。
- 7 ノードの IP アドレスおよびホスト名。
- 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。

4.1.2. クラスタでのノード上の 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>
```

例:

```
$ 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 -l=<pod_selector>
```

例:

```
$ oc describe node --selector=beta.kubernetes.io/os
$ oc describe node -l node-role.kubernetes.io/worker
```

4.1.3. ノードのメモリーと CPU 使用統計の表示

コンテナ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
```

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。

4.2. 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。

4.2.1. 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。



注記

Pod の退避を実行するときは、先にノードをスケジュール対象外としてマークする必要があります。

```
$ oc adm cordon <node1>
NAME      STATUS              ROLES    AGE    VERSION
<node1>   NotReady,SchedulingDisabled worker  1d     v1.13.4+b626c2fe1
```

完了したら、**oc adm uncordon** を使って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>
```

- 以下のコマンドは、1つまたは複数のノードのすべてのまたは選択した Pod を退避します。

```
$ oc adm drain <node1> <node2> [--pod-selector=<pod_selector>]
```

- 以下のコマンドは、**--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、ReplicaSet、ジョブ、DaemonSet、または StatefulSet で管理されていない場合でも削除が実行されます。

```
$ oc adm drain <node1> <node2> --force=true
```

- 以下のコマンドは、**--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
```

- 以下のコマンドは、**true** に設定された **--ignore-daemonsets** フラグを使用して、DaemonSet が管理する Pod を無視します。

```
$ oc adm drain <node1> <node2> --ignore-daemonsets=true
```

- 以下のコマンドは、**--timeout** フラグを使用して、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
```

- 以下のコマンドは、**true** に設定された **--delete-local-data** フラグを使用して、emptyDir 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-local-data=true
```

- 以下のコマンドは、**true** に設定された **--dry-run** オプションを使用して、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。

4.2.2. 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、MachineSet 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 MachineSet に追加されたラベルは、MachineSet 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>
```

例:

```
$ 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>
```

例:

```
$ oc label pods --all status=unhealthy
```

4.2.3. ノードをスケジューリング対象外 (Unschedulable) またはスケジューリング対象 (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リング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リング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リング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リング対象外としてマークします。

```
$ oc adm cordon <node>
```

例:

```
$ oc adm cordon node1.example.com
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リング対象外のノードをスケジューリング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。

4.2.4. 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 を編集します。

1. クラスターにある MachineSet を表示します。

```
$ oc get machinesets -n openshift-machine-api
```

MachineSet は `<clusterid>-worker-<aws-region-az>` の形式で一覧表示されます。

2. MachineSet をスケーリングします。

```
$ oc scale --replicas=2 machineset <machineset> -n openshift-machine-api
```

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について参照してください。

4.2.5. 追加リソース

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 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について](#)」を参照してください。

4.3. ノードの管理

OpenShift Container Platform は、KubeletConfig Custom Resource を使ってノードの設定を管理します。KubeletConfig のインスタンスを作成すると、管理対象の MachineConfig がノードの設定を上書きするように作成されます。



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。

4.3.1. ノードの変更

クラスターまたは MachinePool の設定を変更するには、クラスターリソース定義 (CRD、Cluster Resource Definition) または KubeletConfig インスタンス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している CRD を介して導入された変更を監視します。

手順

1. 設定する必要が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 machineconfigpool の現在のラベルをチェックします。

例:

```
$ oc get machineconfigpool --show-labels
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 machineconfigpool にカスタムラベルを追加します。

例:

```
$ oc label machineconfigpool worker custom-kubelet=enabled
```

2. 設定の変更に KubeletConfig カスタムリソース(CR) を作成します。

例:

custom-config CR の設定例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: custom-config ❶
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: enabled ❷
  kubeletConfig: ❸
    podsPerCore: 10
    maxPods: 250
    systemReserved:
      cpu: 1000m
      memory: 500Mi
    kubeReserved:
      cpu: 1000m
      memory: 500Mi
```

❶ CR に名前を割り当てます。

❷ 設定変更を適用するラベルを指定します。これは、machineconfigpool に追加するラベルになります。

❸ 変更する必要が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>
```

例:

```
$ oc create -f master-kube-config.yaml
```

ほとんどの [KubeletConfig 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

4.4. 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

単一コンテナを保持する Pod は実際には 2 つのコンテナを使用します。2 つ目のコンテナは実際のコンテナ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。

4.4.1. 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. 設定したいノードタイプの静的な Machine Config Pool 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. マシンの Config Pool を表示します。

```
$ oc describe machineconfigpool <name>
```

例:

```
$ oc describe machineconfigpool worker

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、Custom Resource) 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 Machine Config Pool CRD の一覧を示します。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
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
NAME CONFIG UPDATED UPDATING DEGRADED
master master-9cc2c72f205e103bb534 False True False
worker worker-8cecd1236b33ee3f8a5e True False False
```

4.5. NODE TUNING OPERATOR の使用

Node Tuning Operator について説明し、この Operator を使用し、Tuned デーモンのオーケストレーションを実行してノードレベルのチューニングを管理する方法について説明します。

4.5.1. Node Tuning Operator について

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ー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 (現時点でこれはテクノロジープレビュー機能です)。Operator は、コンテナ化された OpenShift Container Platform の Tuned デーモンを Kubernetes DaemonSet 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。

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。

4.5.2. 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
```

4.5.3. 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:** はチューニングされた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。

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:
- match: # optional; if omitted, profile match is assumed unless a profile with a
  higher matches first
  <match> # an optional array
  priority: <priority> # profile ordering priority, lower numbers mean higher priority (0 is the
  highest priority)
  profile: <tuned_profile_name> # e.g. tuned_profile_1
```

```
# ...
- match:
  <match>
  priority: <priority>
  profile: <tuned_profile_name>    # e.g. tuned_profile_n
```

<match> が省略されている場合は、プロファイルの一致 (例: **true**) があることが想定されます。

<match> は、以下のように再帰的に定義されるオプションの配列です。

```
- label: <label_name>    # node or pod label name
  value: <label_value>  # optional node or pod label value; if omitted, the presence of <label_name>
  is enough to match
  type: <label_type>    # optional node or pod type ("node" or "pod"); if omitted, "node" is assumed
  <match>              # an optional <match> array
```

<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> 配列のいずれかの項目が一致する場合、<match> の全体の配列が **true** に評価されます。そのため、配列は論理 OR 演算子として機能します。

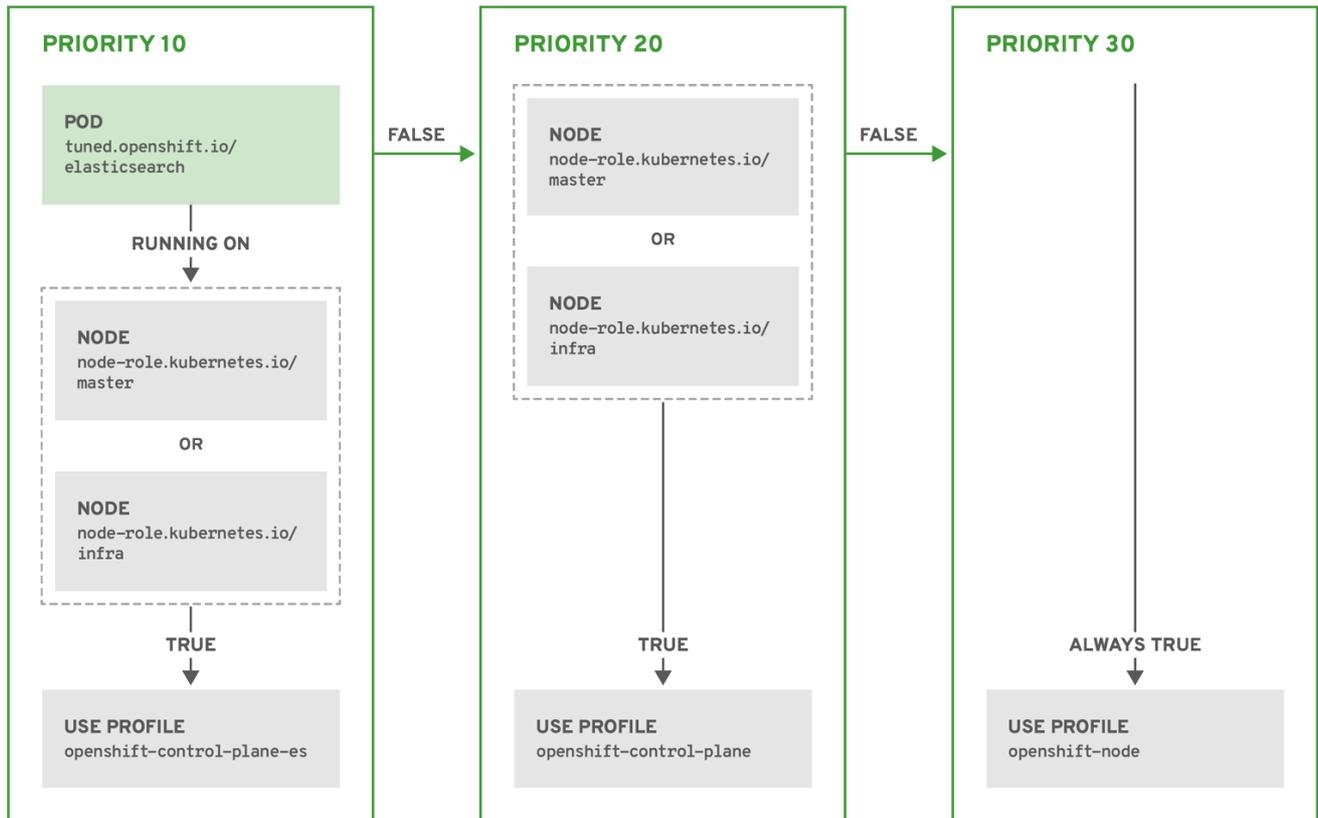
例

```
- 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化された Tuned デーモンは、同じノードに **tuned.openshift.io/elasticsearch** ラベルを持つ 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。



OPENSIFT_10_0319

4.5.4. クラスタに設定されるデフォルトのプロファイル

以下は、クラスタに設定されるデフォルトのプロファイルです。

```
apiVersion: tuned.openshift.io/v1alpha1
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=>131072
      net.netfilter.nf_conntrack_max=1048576
```

```
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
[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
- name: "openshift-control-plane-es"
data: |
[main]
summary=Optimize systems running ES on OpenShift control-plane
include=openshift-control-plane
[sysctl]
vm.max_map_count=262144
- name: "openshift-node-es"
data: |
[main]
summary=Optimize systems running ES on OpenShift nodes
include=openshift-node
[sysctl]
vm.max_map_count=262144
recommend:
- profile: "openshift-control-plane-es"
priority: 10
match:
- label: "tuned.openshift.io/elasticsearch"
```

```
  type: "pod"
  match:
  - label: "node-role.kubernetes.io/master"
  - label: "node-role.kubernetes.io/infra"

- profile: "openshift-node-es"
  priority: 20
  match:
  - label: "tuned.openshift.io/elasticsearch"
    type: "pod"

- profile: "openshift-control-plane"
  priority: 30
  match:
  - label: "node-role.kubernetes.io/master"
  - label: "node-role.kubernetes.io/infra"

- profile: "openshift-node"
  priority: 40
```

4.5.5. 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、「[Available Tuned Plug-ins](#)」および「[Getting Started with Tuned](#)」を参照してください。

4.6. 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。

4.6.1. インフラストラクチャーノードの再起動について

インフラストラクチャーノードは、OpenShift Container Platform 環境の各部分を実行するようにラベル付けされたノードです。現在、ノードの再起動を管理する最も簡単な方法は、インフラストラクチャーを実行するための少なくとも3つのノードを利用可能な状態にすることです。インフラストラクチャーを実行するノードは、**マスターノード**と呼ばれます。

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 OpenShift Container Platform で実行されているアプリケーションのサービス中断を引き起こす可能性がある一般的なエラーを示しています。

- ノード A がスケジューリング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。これは、ノード B が両方のレジストリー Pod を実行していることを意味します。
- ノード B はスケジューリング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の3つのマスターノードを使用する同じプロセスではサービス中断が生じません。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た最後のノードはゼロ (0) レジストリーを実行したままになります。他の2つのノードは2つのレジストリーと1つのレジストリーをそれぞれ実行します。最適なソリューションとして、Pod の非アフィニティーを使用します。

4.6.2. 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この Pod を `unready` (準備ができていない) 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: ❶
      preferredDuringSchedulingIgnoredDuringExecution: ❷
      - weight: 100 ❸
        podAffinityTerm:
          labelSelector:
            matchExpressions:
              - key: registry ❹
                operator: In ❺
                values:
                  - default
          topologyKey: kubernetes.io/hostname
  
```

- ❶ Pod の非アフィニティーを設定するためのスタンザです。
- ❷ `preferred` (優先) ルールを定義します。
- ❸ `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せん。
- ❹ 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。
- ❺ 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。

4.6.3. 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。

4.7. 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。

- コンテナのガベージコレクション: 終了したコンテナを削除します。
- イメージのガベージコレクション: 実行中のどの Pod から参照されていないイメージを削除します。

4.7.1. 終了したコンテナ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、コンテナも削除されます。コンテナは 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。

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。

4.7.2. ガベージコレクションによってイメージが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4.2 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
imageMinimumGauge	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。

4.7.3. コンテナ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の Config Pool 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、各マシンの Config Pool 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。

- コンテナのソフトエビクション
- コンテナのハードエビクション
- イメージのエビクション

ソフトコンテナエビクションの場合は、エビクションされるまでの猶予期間を設定することもできます。

前提条件

1. 設定したいノードタイプの静的な Machine Config Pool 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. マシンの Config Pool を表示します。

```
$ oc describe machineconfigpool <name>
```

例:

```
$ oc describe machineconfigpool worker

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、Custom Resource) を作成します。

コンテナ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
    evictionSoft:
      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:
  memory.available: "200Mi"
  nodefs.available: "5%"
  nodefs.inodesFree: "4%"
  imagefs.available: "10%"
  imagefs.inodesFree: "5%"
evictionPressureTransitionPeriod: 0s 6
imageMinimumGCAge: 5m 7
imageGCHighThresholdPercent: 80 8
imageGCLowThresholdPercent: 75 9

```

- 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** および **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 エビクションの不足状態から移行するまでの待機時間
- 7 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。
- 8 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。
- 9 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
```

例:

```

oc create -f gc-container.yaml
kubenetconfig.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' と表示されません。

```

$ oc get machineconfigpool

NAME      CONFIG                                UPDATED  UPDATING

```

```

master rendered-master-546383f80705bd5aeaba93 True False
worker rendered-worker-b4c51bb33ccea6fc4a6a5 False True

```

4.8. OPENSIFT CONTAINER PLATFORM クラスタ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、各ノードは、ホスト上のすべての基礎となるノードコンポーネント (kubelet、kube-proxy など) および残りのシステムコンポーネント (sshd、NetworkManager など) に使用されるリソースの一部を予約できます。指定されると、スケジューラーは、ノードが Pod に割り当てたリソース (例: メモリー、CPU) についての詳細な情報を取得できます。

4.8.1. 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	ノードコンポーネント用に予約されたリソースです。デフォルトは none です。
system-reserved	残りのシステムコンポーネント用に予約されたリソースです。デフォルトは none です。

フラグが設定されていない場合、これはデフォルトで 0 になります。フラグのいずれ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。

4.8.1.1. OpenShift Container Platform による割り当てられたリソースの計算方法

リソースの割り当てられる量は以下の数式に基づいて計算されます。

$$[\text{Allocatable}] = [\text{Node Capacity}] - [\text{kube-reserved}] - [\text{system-reserved}] - [\text{Hard-Eviction-Thresholds}]$$



注記

Hard-Eviction-Thresholds を割り当て可能なリソースから差し引く点はシステムの信頼性を強化するための動作の変更点です。この値に基づいて割り当て可能なリソースをノードレベルでエンドユーザー Pod に対して適用できます。 **experimental-allocatable-ignore-eviction** 設定は、レガシー動作を保持するために利用できますが、今後のリリースでは非推奨となります。

[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** および **--kube-reserved** の設定をより容易に行えるようにするには、ノード要約 API を使用して対応するノードのリソース使用状況をインストロスペクトできます。この API は `<master>/api/v1/nodes/<node>/proxy/stats/summary` からアクセスできます。

4.8.1.2. 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 がリソースのシステムサービス (例: コンテナランタイム、ノードエージェントなど) のリソース不足の状態を防ぎ、ノードの信頼性を大幅に強化します。管理者に対しては、ノードの信頼性を強化するために必要なノードの使用状況のターゲットに基づいてリソースを予約することが強く奨励されます。

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。

オプションとして、`enforce-node-allocatable` フラグでトークンを指定することで、ノードが kube で予約される制限およびシステムで予約される制限を実施するようにできます。これらが指定される場合、対応する **--kube-reserved-cgroup** または **--system-reserved-cgroup** を指定する必要があります。今後のリリースでは、ノードおよびコンテナランタイムは **system.slice** と切り離された共通の cgroup でパッケージ化されます。パッケージ化されるまでは、ユーザーは `enforce-node-allocatable` フラグのデフォルト値を変更しないようにすることを推奨します。

管理者は Guaranteed 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予想外の動作をする可能性があります、この動作はクラスター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。システム予約の制限を実施することにより、ノード上の重要なシステムサービスで CPU が不足したり、OOM による強制終了が生じる可能性があります。そのため、オペレーターが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で OOM による強制終了が発生した場合に確実に復元できる場合にのみシステム予約の制限を実施することが推奨されます。

したがって、ユーザーはデフォルトで **Pods** に対してのみノードの割り当て可能分を実施し、ノード全体の信頼性を維持するためにシステムデーモンの適切な予約を確保しておくことを強くお勧めします。

4.8.1.3. 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。システムデーモンによるメモリーの使用量がメモリーの予約されている量を超えていると、OOM イベントが発生し、ノード全体やノードで実行されているすべての Pod に影響が及び可能性があります。システムの OOM を防止する (またはその発生可能性を下げる) ために、ノードは Out Of Resource Handling (リソース不足の処理) 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**
- `--kube reserved`: **2Gi**
- `--system-reserved`: **1Gi**
- `--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**) を実施する場合、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。

4.8.1.4. 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。

4.8.2. 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。管理者が一時ストレージのテクノロジープレビューを有効にしている場合、**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。

前提条件

1. **--system-reserved** および **--kube-reserved** の設定を決定しやすくするには、ノード要約 API を使用して対応するノードのリソース使用状況をインストロスペクトできます。この API は **<master>/api/v1/nodes/<node>/proxy/stats/summary** からアクセスできます。ノードについて以下のコマンドを実行します。

```
$ curl <certificate details> https://<master>/api/v1/nodes/<node-name>/proxy/stats/summary
```

REST API Overview には、証明書の詳細情報が記載されています。

たとえば、**cluster.node22** ノードからリソースにアクセスするには、以下の手順を実行できます。

```
$ curl <certificate details> https://<master>/api/v1/nodes/cluster.node22/proxy/stats/summary
{
  "node": {
    "nodeName": "cluster.node22",
    "systemContainers": [
      {
        "cpu": {
          "usageCoreNanoSeconds": 929684480915,
          "usageNanoCores": 190998084
```

```

    },
    "memory": {
      "rssBytes": 176726016,
      "usageBytes": 1397895168,
      "workingSetBytes": 1050509312
    },
    "name": "kubelet"
  },
  {
    "cpu": {
      "usageCoreNanoSeconds": 128521955903,
      "usageNanoCores": 5928600
    },
    "memory": {
      "rssBytes": 35958784,
      "usageBytes": 129671168,
      "workingSetBytes": 102416384
    },
    "name": "runtime"
  }
]
}
}

```

2. 設定したいノードタイプの静的な Machine Config Pool 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. マシンの Config Pool を表示します。

```
$ oc describe machineconfigpool <name>
```

例:

```

$ oc describe machineconfigpool worker

apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  creationTimestamp: 2019-02-08T14:52:39Z
  generation: 1
  labels:
    custom-kubelet: small-pods ❶

```

- ❶ 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、Custom Resource) 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 ❶
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: small-pods ❷
  kubeletConfig:
    systemReserved:
      cpu: 500m
      memory: 512Mi
    kubeReserved:
      cpu: 500m
      memory: 512Mi

```

4.9. ノード監査ログの表示

監査は、システムに影響を与えた一連のアクティビティを個別のユーザー、管理者その他システムのコンポーネント別に記述したセキュリティー関連の時系列のレコードを提供します。

4.9.1. API の監査ログについて

監査は API サーバーレベルで実行され、サーバーに送られるすべての要求をログに記録します。それぞれの監査ログには、以下の情報が含まれます。

表4.3 監査ログフィールド

フィールド	説明
level	イベントが生成された監査レベル。
auditID	要求ごとに生成される一意の監査 ID。
stage	このイベントインスタンスの生成時の要求処理のステージ。
requestURI	クライアントによってサーバーに送信される要求 URI。
verb	要求に関連付けられる Kubernetes の動詞。リソース以外の要求の場合、これは小文字の HTTP メソッドになります。
user	認証されたユーザーの情報。
impersonatedUser	オプション。偽装ユーザーの情報 (要求で別のユーザーを偽装する場合)。
sourceIPs	オプション。要求の送信元および中間プロキシからのソース IP。

フィールド	説明
userAgent	オプション。クライアントが報告するユーザーエージェントの文字列。ユーザーエージェントはクライアントによって提供されており、信頼できないことに注意してください。
objectRef	オプション。この要求のターゲットとなっているオブジェクト参照。これは、 List タイプの要求やリソース以外の要求には適用されません。
responseStatus	オプション。 responseObject が Status タイプでなくても設定される応答ステータス。正常な応答の場合、これにはコードのみが含まれます。ステータス以外のタイプのエラー応答の場合、これにはエラーメッセージが自動的に設定されます。
requestObject	オプション。JSON 形式の要求からの API オブジェクト。 requestObject は、バージョンの変換、デフォルト設定、受付またはマージの前に要求の場合のように記録されます (JSON として再エンコードされる可能性がある)。これは外部のバージョン付けされたオブジェクトタイプであり、それ自体では有効なオブジェクトではない可能性があります。これはリソース以外の要求の場合には省略され、要求レベル以上でのみログに記録されます。
responseObject	オプション。JSON 形式の応答で返される API オブジェクト。 responseObject は外部タイプへの変換後に記録され、JSON としてシリアル化されます。これはリソース以外の要求の場合には省略され、応答レベルでのみログに記録されます。
requestReceivedTimestamp	要求が API サーバーに到達した時間。
stageTimestamp	要求が現在の監査ステージに達した時間。
annotations	オプション。監査イベントと共に保存される構造化されていないキーと値のマッピング。これは、認証、認可、受付プラグインなど、要求提供チェーンで呼び出されるプラグインによって設定される可能性があります。これらのアノテーションは監査イベント用のもので、送信されたオブジェクトの metadata.annotations に対応しないことに注意してください。キーは、名前の競合が発生しないように通知コンポーネントを一意に識別する必要があります (例: podsecuritypolicy.admission.k8s.io/policy)。値は短くする必要があります。アノテーションはメタデータレベルに含まれます。

Kubernetes API サーバーの出力例:

```
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "level": "Metadata",
  "auditID": "ad209ce1-fec7-4130-8192-c4cc63f1d8cd",
  "stage": "ResponseComplete",
  "requestURI": "/api/v1/namespaces/openshift-kube-controller-manager/configmaps/cert-recovery-controller-lock?timeout=35s",
  "verb": "update",
  "user": {
    "username": "system:serviceaccount:openshift-kube-controller-manager:localhost-recovery-client",
    "uid": "dd4997e3-d565-4e37-80f8-7fc122ccd785",
    "groups": [
      "system:serviceaccounts",
      "system:serviceaccounts:openshift-kube-controller-manager",
      "system:authenticated"
    ],
    "sourceIPs": ["::1"],
    "userAgent": "cluster-kube-controller-manager-operator/v0.0.0 (linux/amd64) kubernetes/$Format",
    "objectRef":
```

```
{
  "resource": "configmaps",
  "namespace": "openshift-kube-controller-manager",
  "name": "cert-recovery-controller-lock",
  "uid": "5c57190b-6993-425d-8101-8337e48c7548",
  "apiVersion": "v1",
  "resourceVersion": "574307",
  "responseStatus": {
    "metadata": {
      "code": 200,
      "requestReceivedTimestamp": "2020-04-02T08:27:20.200962Z",
      "stageTimestamp": "2020-04-02T08:27:20.206710Z",
      "annotations": {
        "authorization.k8s.io/decision": "allow",
        "authorization.k8s.io/reason": "RBAC: allowed by ClusterRoleBinding \"system:openshift:operator:kube-controller-manager-recovery\" of ClusterRole \"cluster-admin\" to ServiceAccount \"localhost-recovery-client/openshift-kube-controller-manager\""
      }
    }
  }
}
```

4.9.2. 監査ログの表示

それぞれのマスターノードについて OpenShift Container Platform API サーバーまたは Kubernetes API サーバーのログを表示することができます。

手順

監査ログを表示するには、以下を実行します。

1. OpenShift Container Platform API サーバーログを表示します。
 - a. 必要に応じて、表示する必要のあるログのノード名を取得します。

```
$ oc adm node-logs --role=master --path=openshift-apiserver/
ip-10-0-140-97.ec2.internal audit-2019-04-09T00-12-19.834.log
ip-10-0-140-97.ec2.internal audit-2019-04-09T11-13-00.469.log
ip-10-0-140-97.ec2.internal audit.log
ip-10-0-153-35.ec2.internal audit-2019-04-09T00-11-49.835.log
ip-10-0-153-35.ec2.internal audit-2019-04-09T11-08-30.469.log
ip-10-0-153-35.ec2.internal audit.log
ip-10-0-170-165.ec2.internal audit-2019-04-09T00-13-00.128.log
ip-10-0-170-165.ec2.internal audit-2019-04-09T11-10-04.082.log
ip-10-0-170-165.ec2.internal audit.log
```

- b. 特定のマスターノードとタイムスタンプの OpenShift Container Platform API サーバーのログを表示したり、そのマスターのすべてのログを表示します。

```
$ oc adm node-logs <node-name> --path=openshift-apiserver/<log-name>
```

例:

```
$ oc adm node-logs ip-10-0-140-97.ec2.internal --path=openshift-apiserver/audit-2019-04-08T13-09-01.227.log
$ oc adm node-logs ip-10-0-140-97.ec2.internal --path=openshift-apiserver/audit.log
```

以下のような出力が表示されます。

```
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "level": "Metadata",
  "auditID": "ad209ce1-fec7-4130-8192-c4cc63f1d8cd",
  "stage": "ResponseComplete",
  "requestURI": "/api/v1/namespaces/openshift-kube-controller-manager/configmaps/cert-recovery-controller-lock?timeout=35s",
  "verb": "update",
  "user": {
    "username": "system:serviceaccount:openshift-kube-controller-manager:localhost-recovery-client",
    "uid": "dd4997e3-d565-4e37-80f8-7fc122ccd785",
    "groups": [
      "system:serviceaccounts",
      "system:serviceaccounts:openshift-kube-controller-manager",
      "system:authenticated"
    ],
    "sourceIPs":
  }
}
```

```
["::1"],"userAgent":"cluster-kube-controller-manager-operator/v0.0.0 (linux/amd64)
kubernetes/$Format","objectRef":{"resource":"configmaps","namespace":"openshift-kube-
controller-manager","name":"cert-recovery-controller-lock","uid":"5c57190b-6993-425d-
8101-8337e48c7548","apiVersion":"v1","resourceVersion":"574307"},"responseStatus":
{"metadata":{},"code":200},"requestReceivedTimestamp":"2020-04-
02T08:27:20.200962Z","stageTimestamp":"2020-04-
02T08:27:20.206710Z","annotations":
{"authorization.k8s.io/decision":"allow","authorization.k8s.io/reason":"RBAC: allowed by
ClusterRoleBinding \"system:openshift:operator:kube-controller-manager-recovery\" of
ClusterRole \"cluster-admin\" to ServiceAccount \"localhost-recovery-client/openshift-
kube-controller-manager\""}}}
```

2. Kubernetes API サーバーログを表示します。

- a. 必要に応じて、表示する必要があるログのノード名を取得します。

```
$ oc adm node-logs --role=master --path=kube-apiserver/

ip-10-0-140-97.ec2.internal audit-2019-04-09T14-07-27.129.log
ip-10-0-140-97.ec2.internal audit-2019-04-09T19-18-32.542.log
ip-10-0-140-97.ec2.internal audit.log
ip-10-0-153-35.ec2.internal audit-2019-04-09T19-24-22.620.log
ip-10-0-153-35.ec2.internal audit-2019-04-09T19-51-30.905.log
ip-10-0-153-35.ec2.internal audit.log
ip-10-0-170-165.ec2.internal audit-2019-04-09T18-37-07.511.log
ip-10-0-170-165.ec2.internal audit-2019-04-09T19-21-14.371.log
ip-10-0-170-165.ec2.internal audit.log
```

- b. 特定のマスターノードとタイムスタンプの Kubernetes API サーバーログを表示したり、そのマスターのすべてのログを表示します。

```
$ oc adm node-logs <node-name> --path=kube-apiserver/<log-name>
```

例:

```
$ oc adm node-logs ip-10-0-140-97.ec2.internal --path=kube-apiserver/audit-2019-04-
09T14-07-27.129.log
$ oc adm node-logs ip-10-0-170-165.ec2.internal --path=kube-apiserver/audit.log
```

以下のような出力が表示されます。

```
{"kind":"Event","apiVersion":"audit.k8s.io/v1","level":"Metadata","auditID":"ad209ce1-fec7-
4130-8192-
c4cc63f1d8cd","stage":"ResponseComplete","requestURI":"/api/v1/namespaces/openshift-
kube-controller-manager/configmaps/cert-recovery-controller-lock?
timeout=35s","verb":"update","user":{"username":"system:serviceaccount:openshift-kube-
controller-manager:localhost-recovery-client","uid":"dd4997e3-d565-4e37-80f8-
7fc122ccd785","groups":["system:serviceaccounts","system:serviceaccounts:openshift-
kube-controller-manager","system:authenticated"]},"sourceIPs":
["::1"],"userAgent":"cluster-kube-controller-manager-operator/v0.0.0 (linux/amd64)
kubernetes/$Format","objectRef":{"resource":"configmaps","namespace":"openshift-kube-
controller-manager","name":"cert-recovery-controller-lock","uid":"5c57190b-6993-425d-
8101-8337e48c7548","apiVersion":"v1","resourceVersion":"574307"},"responseStatus":
{"metadata":{},"code":200},"requestReceivedTimestamp":"2020-04-
02T08:27:20.200962Z","stageTimestamp":"2020-04-
```

```
02T08:27:20.206710Z", "annotations":  
{ "authorization.k8s.io/decision": "allow", "authorization.k8s.io/reason": "RBAC: allowed by  
ClusterRoleBinding \"system:openshift:operator:kube-controller-manager-recovery\" of  
ClusterRole \"cluster-admin\" to ServiceAccount \"localhost-recovery-client/openshift-  
kube-controller-manager\""} }
```

第5章 コンテナの使用

5.1. コンテナについて

OpenShift Container Platform アプリケーションの基本的な単位は **コンテナ** と呼ばれています。Linux **コンテナテクノロジー** 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で実行される可能性があります。通常、コンテナは任意のワークロードに使用されますが、各コンテナは 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のオーケストレーションを実行する機能を追加します。

5.2. POD のデプロイ前の、INIT コンテナの使用によるタスクの実行

OpenShift Container Platform は、**Init コンテナ** を提供します。このコンテナは、アプリケーションコンテナの前に実行される特殊なコンテナであり、アプリのイメージに存在しないユーティリティ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。

5.2.1. Init コンテナについて

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に加えて、init コンテナを持つことができます。Init コンテナにより、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 コンテナは以下のことを行うことができます。

- セキュリティ上の理由のためにアプリケーションコンテナイメージに含めることが望ましくないユーティリティ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 namespaces を使用して、アプリケーションコンテナ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 コンテナは、次のコンテナ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=$(hostname) & ip=$(ip addr show | grep ^eth | awk '{print $2}' | sed 's/\/.*$//')
```

- **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が起動するまでしばらく待機します。
- Git リポジトリ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。

5.2.2. Init コンテナの作成

以下の例は、2 つの init コンテナを持つ単純な 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。両方のコンテナが完了すると、Pod が開始されます。

手順

1. init コンテナの 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: busybox
      command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
    - name: init-myservice
      image: busybox
      command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
    - name: init-mydb
      image: busybox
      command: ['sh', '-c', 'until nslookup 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

pod/myapp-pod created
```

5. Pod のステータスを表示します。

```
$ oc get pods
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
NAME                READY   STATUS    RESTARTS   AGE
myapp-pod           1/1     Running   0           2m
```

5.3. ボリュームの使用によるコンテナデータの永続化

コンテナ内のファイルは一時的なものです。そのため、コンテナ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。

5.3.1. 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。

5.3.2. 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 DeploymentConfig のような 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 コマンド内の **object_selection** に、以下のいずれかを指定します。

表5.1 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 コマンドに **operation** の **--add**、**--remove**、または **--list** を指定します。

必須パラメーター

いずれの **<mandatory_parameters>** 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 **<options>** 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。

5.3.3. 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> --list [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。	**

例:

- 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 --list
```

- すべての DeploymentConfig 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
```

5.3.4. 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5.2 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。	'*'
-m, --mount-path	選択されたコンテナ内のマウントパス。	
--path	ホストパス。 --type=hostPath の必須パラメーターです。	
--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	Persistent Volume Claim (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。	
--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

例:

- 新規ボリュームソース `emptyDir` をデプロイメント設定の `レジストリー` に追加するには、以下の手順を実行します。

```
$ 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 `c1` の `/data` にマウントし、サーバー上でデプロイメント設定を更新するには、以下の手順を実行します。

```
$ 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 `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"
  } }'
```

5.3.5. 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修正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]
```

例:

- 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 Persistent Volume Claim (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 `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
```

5.3.6. 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5.3 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

例:

- DeploymentConfig **d1** からボリューム **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
```

- DeploymentConfig **d1** のコンテナ **c1** からボリューム **v1** をアンマウントし、**d1** のコンテナ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
```

5.3.7. Pod 内での複数の用途のためのボリュームの設定

ボリュームを設定して、単一 Pod で複数の使用目的のために1つのボリュームを共有できます。**volumeMounts.subPath** プロパティを使用し、ボリュームのルートの代わりにボリューム内で **subPath** を指定します。

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>
sh-4.2$ ls /path/to/volume/subpath/mount
example_file1 example_file2 example_file3
```

2. **subPath** を指定します。

subPath の使用例

```
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 1 データベースは **mysql** フォルダに保存されます。

2 2 HTML コンテンツは **html** フォルダに保存されます。

5.4. 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。

5.4.1. 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、configmap、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、configmap、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。

以下の一般的なシナリオは、展開されるプロジェクトを使用する方法について示しています。

ConfigMap、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 OpenStack を Kubernetes にデプロイす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る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 OpenStack 設定を生成できます。

ConfigMap + 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をデプロイできます。たとえば、configmap 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、オペレーター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。

5.4.1.1. 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 configmap を含む Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
```

```
volumeMounts: ❶
- name: all-in-one
  mountPath: "/projected-volume" ❷
  readOnly: true ❸
volumes: ❹
- name: all-in-one ❺
  projected:
    defaultMode: 0400 ❻
    sources:
      - secret:
          name: mysecret ❼
          items:
            - key: username
              path: my-group/my-username ❽
      - downwardAPI: ❾
          items:
            - path: "labels"
              fieldRef:
                fieldPath: metadata.labels
            - path: "cpu_limit"
              resourceFieldRef:
                containerName: container-test
                resource: limits.cpu
      - configMap: ❿
          name: myconfigmap
          items:
            - key: config
              path: my-group/my-config
              mode: 0777 ⓫
```

- ❶ シークレットを必要とする各コンテナの **volumeMounts** セクションを追加します。
- ❷ 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。
- ❸ **readOnly** を **true** に設定します。
- ❹ 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。
- ❺ ボリュームの名前を指定します。
- ❻ ファイルに実行パーミッションを設定します。
- ❼ 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。
- ❽ **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-config** に置かれます。
- ❾ Downward API ソースを追加します。
- ❿ ConfigMap ソースを追加します。
- ⓫ 特定の展開におけるモードを設定します。



注記

Pod に複数のコンテナがある場合、それぞれのコンテナには **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。

5.4.1.2. 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。

5.4.2. 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、「**Projected 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: MWYyZDFIMmU2N2Rm
  user: YWRtaW4=
```

user および **pass** の値には、**base64** でエンコーディングされた任意の有効な文字列を使用できます。ここで使用される例は base64 でエンコーディングされた値 (**user: admin**、**pass:1f2d1e2e67df**) になります。

```
$ echo -n "admin" | base64
YWRtaW4=
$ echo -n "1f2d1e2e67df" | base64
MWYyZDFIMmU2N2Rm
```

- 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>
```

例:

```
$ oc create -f secret.yaml
secret "mysecret" created
```

- 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>
$ oc get secret <secret-name> -o yaml
```

例:

```
$ oc get secret mysecret
NAME      TYPE      DATA   AGE
myscret  Opaque    2       17h
```

```
$ oc get secret mysecret -o yaml
```

```
apiVersion: v1
data:
  pass: MWYyZDFIMmU2N2Rm
  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
```

- 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: ❶
        name: user
      - secret: ❷
        name: pass

```

❶ ❷ 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
```

例:

```
$ oc create -f secret-pod.yaml
pod "test-projected-volume" created
```

6. Pod コンテナ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>
```

出力は以下のようになります。

```
$ oc get pod test-projected-volume
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、実行中のコンテナに対してシェルを開きます。

```
$ oc exec -it <pod> <command>
```

例:

```
$ 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
bin      home      root      tmp
dev      proc      run       usr
etc      projected-volume sys       var

```

5.5. コンテナによる 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。

5.5.1. Downward API の使用によるコンテナへの 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。

5.5.2. Downward API を使用してコンテナの値を使用する方法について

コンテナは、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。

5.5.2.1. 環境変数の使用によるコンテナ値の使用

コンテナの環境変数を使用する際に、**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
```

5.5.2.2. ボリュームプラグインを使用したコンテナ値の使用

コンテナは、ボリュームプラグインを使用して API 値を使用できます。

コンテナは、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
cluster=downward-api-test-cluster1
rack=rack-123
zone=us-east-coast
annotation1=345
annotation2=456
kubernetes.io/config.source=api

```

5.5.3. Downward API を使用してコンテナ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。

5.5.3.1. 環境変数を使用したコンテナリソースの使用

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
```

5.5.3.2. ボリュームプラグインを使用したコンテナリソースの使用

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
```

5.5.4. 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

```

- secret.yaml ファイルから **Secret** を作成します。

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

- 上記の **Secret** 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
```

- pod.yaml** ファイルから Pod を作成します。

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

- コンテナのログで **MY_SECRET_USERNAME** の値を確認します。

```
$ oc logs -p dapi-env-test-pod
```

5.5.5. 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

- configmap.yaml** ファイルを作成します。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfigmap
data:
  mykey: myvalue
```

- configmap.yaml** ファイルから **ConfigMap** を作成します。

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

- 上記の **ConfigMap** 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. コンテナのログで **MY_CONFIGMAP_VALUE** の値を確認します。

```
$ oc logs -p dapi-env-test-pod
```

5.5.6. 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. コンテナのログで **MY_ENV_VAR_REF_ENV** 値を確認します。

```
$ oc logs -p dapi-env-test-pod
```

5.5.7. 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. コンテナのログで **MY_NEW_ENV** 値を確認します。

```
$ oc logs -p dapi-env-test-pod
```

5.6. OPENSIFT CONTAINER PLATFORM コンテナへの/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。

5.6.1. 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>]
```

5.6.1.1. 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は 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。

5.6.2. コンテナへの/からのファイルのコピー

コンテナへの/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 sync 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 **rsync** と同様に機能する訳ではありません。たとえば、**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>
```

例:

```
$ oc rsync /home/user/source devpod1234:/src
```

```
WARNING: cannot use rsync: rsync not available in container
status.txt
```

- 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
```

```
oc rsync devpod1234:/src/status.txt /home/user/
WARNING: cannot use rsync: rsync not available in container
status.txt
```

5.6.3. 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場合 (**--exclude-from=FILE** オプションなど)、以下のように標準の **rsync** の **--rsh (-e)** オプションまたは **RSYNC_RSH** 環境変数を回避策として使用することができます。

```
$ rsync --rsh='oc rsh' --exclude-from=FILE SRC POD:DEST
```

または、以下を実行します。

```
$ export RSYNC_RSH='oc rsh'
$ 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。

5.7. OPENSIFT CONTAINER PLATFORM コンテナでのリモートコマンドの実行

OpenShift Container Platform コンテナ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。

5.7.1. コンテナでのリモートコマンドの実行

リモートコンテナ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>]
```

例:

```
$ oc exec mypod date
Thu Apr 9 02:21:53 UTC 2015
```



重要

セキュリティ保護の理由により、**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、特権付きコンテナにアクセスしようとしても機能しません。

5.7.2. 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の名前です。
- **<command>** は実行される必要なコマンドです。

例:

```
/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 **SPDY** 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。

5.8. コンテナ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。

5.8.1. 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な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、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。

5.8.2. 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>]
```

例:

- 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
```

```
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
```

```
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
```

```
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
```

5.8.3. 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。

例:

```
/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 **SPDY** 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。

第6章 クラスターの操作

6.1. OPENSIFT 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。

6.1.1. 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。

6.1.2. 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
```

- プロジェクトの名前。

例:

```
$ oc get events -n openshift-config
```

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" iface 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。
 - 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。

6.1.3. 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6.1 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6.2 コンテナイベント

名前	説明
BackOff	バックオフ (再起動) によりコンテナが失敗しました。
Created	コンテナが作成されました。
Failed	プル/作成/起動が失敗しました。
Killing	コンテナを強制終了しています。
Started	コンテナが起動しました。
Preempting	他の Pod を退避します。
ExceededGrace Period	コンテナランタイムは、指定の猶予期間以内に Pod を停止しませんでした。

表6.3 正常性イベント

名前	説明
Unhealthy	コンテナが正常ではありません。

表6.4 イメージイベント

名前	説明
BackOff	バックオフ (コンテナ起動、イメージのプル)。

名前	説明
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6.5 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6.6 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
InsufficientFreeCPU	空き CPU が十分にありません。
InsufficientFreeMemory	空きメモリーが十分にありません。
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。

名前	説明
NodeNotSchedulable	ノードがスケジュール可能ではありません。
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。

名前	説明
SuccessfulDetachVolume	ボリュームの割り当てが正常に解除されました。
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6.7 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6.8 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6.9 Pod 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6.10 Horizontal Pod AutoScaler 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6.11 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ースがなくなり、Pod が停止します。

表6.12 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6.13 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6.14 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6.15 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。
IngressIPRangeFull	サービスに割り当てる Ingress IP がありません。

表6.16 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6.17 DaemonSet 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6.18 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。
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>

名前	説明
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。

6.2. OPENSIFT 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。

6.2.1. 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。

6.2.2. 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

- **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。
- 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

コマンドラインでツールを実行するには、以下を実行します。

1. 次のコマンドを実行します。

```
$ ./cluster-capacity --kubeconfig <path-to-kubeconfig> \ ①
--podspec <path-to-pod-spec> ②
```

- ① Kubernetes 設定ファイルへのパスを指定します。
- ② サンプル Pod 仕様ファイルへのパスを指定します。

--verbose オプションを追加して、クラスター内の各ノードにスケジュールできる Pod 数についての詳細説明を出力できます。

```
$ ./cluster-capacity --kubeconfig <path-to-kubeconfig> \
  --podspect <path-to-pod-spec> --verbose
```

2. 以下のような出力を表示します。

```
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)
```

上記の例では、クラスターにスケジュールできる Pod の見積り数は 52 です。

6.2.3. 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** 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 -
kind: ClusterRole
apiVersion: v1
metadata:
  name: cluster-capacity-role
rules:
- apiGroups: ["" ]
  resources: ["pods", "nodes", "persistentvolumeclaims", "persistentvolumes", "services"]
  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. クラスタ容量分析は、**cluster-capacity-configmap** という名前の **ConfigMap** 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 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** クラスタ容量ツールにクラスタ内で 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
small-pod pod requirements:
- CPU: 150m
- Memory: 100Mi

```

The cluster can schedule 52 instance(s) of the pod small-pod.

Termination reason: Unscheduleable: 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)

```

6.3. コンテナメモリーとリスク要件を満たすためのクラスタメモリーの設定

クラスタ管理者は、以下を実行し、クラスタ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メモリーパラメーターを設定する
- コンテナ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

- コンテナでの実行に関連するメモリ関連のエラー状態を診断し、これを解決する

6.3.1. アプリケーションメモリの管理について

まず OpenShift Container Platform によるコンピュータ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、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 の各コンテナに設定できます。

メモリ要求とメモリ制限について、以下の点に注意してください。

- **メモリ要求**
 - メモリ要求値は、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、コンテナのノードへのスケジュール時にメモリ要求を考慮し、コンテナの使用のために選択されたノードで要求されたメモリをフェンスオフします。
 - ノードのメモリが使い切られると、OpenShift Container Platform はメモリ使用がメモリ要求を最も超過しているコンテナのエビクションを優先します。メモリ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要求の値を上書きできます。
- **メモリ制限**
 - メモリ制限値が指定されている場合、コンテナのすべてのプロセスに割り当て可能なメモリにハード制限を指定します。
 - コンテナのすべてのプロセスで割り当てられるメモリがメモリ制限を超過する場合、ノードの OOM killer はコンテナのプロセスをすぐに選択し、これを強制終了します。
 - メモリ要求とメモリ制限の両方が指定される場合、メモリ制限の値はメモリ要求の値よりも大きいのか、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。

6.3.1.1. 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のメモリ使用の判別**

必要時に予想される平均およびピーク時のコンテナのメモリ使用を判別します (例: 別の負荷テストを実行)。コンテナ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. コンテナのメモリー要求の設定

上記に基づいてコンテナ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. コンテナのメモリー制限の設定 (必要な場合)

必要時にコンテナ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。

6.3.2. OpenShift Container Platform の OpenJDK 設定について

デフォルトの OpenJDK 設定はコンテナ化された環境では機能しません。そのため、コンテナで 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。

6.3.2.1. 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タ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. コンテナのメモリ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。
これにより、**-XX:MaxRAM** がコンテナのメモリ制限に設定され、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。

6.3.2.2. JVM で未使用メモリ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が使用中のメモリ (**-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。

6.3.2.3. コンテナ内のすべての 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"`
```

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。

6.3.3. 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。

1. Pod を作成します。

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

2. リモートシェルを使用して Pod にアクセスします。

```
$ oc rsh test
```

3. 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
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内から読み取ることもできます。

6.3.4. OOM の強制終了ポリシーについて

OpenShift Container Platform は、コンテナ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のプロセスを強制終了する場合があります。

プロセスが OOM によって強制終了される場合、コンテナがすぐに終了する場合もあれば、終了しない場合もあります。コンテナの PID1 プロセスが **SIGKILL** を受信する場合、コンテナはすぐに終了します。それ以外の場合、コンテナ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. `/sys/fs/cgroup/memory/memory.oom_control` の `oom_kill` カウンターの増分が確認されません。

```
$ grep '^oom_kill' /sys/fs/cgroup/memory/memory.oom_control
oom_kill 0
$ sed -e " </dev/zero # provoke an OOM kill
Killed
$ echo $?
137
$ 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、ReplicationController 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。

再起動されない場合、Pod のステータスは以下ようになります。

```
$ oc get pod test
NAME    READY   STATUS    RESTARTS   AGE
test    0/1     OOMKilled 0           1m

$ oc get pod test -o yaml
...
status:
  containerStatuses:
  - name: test
    ready: false
    restartCount: 0
```

```

state:
  terminated:
    exitCode: 137
    reason: OOMKilled
phase: Failed

```

再起動される場合、そのステータスは以下ようになります。

```

$ oc get pod test
NAME    READY   STATUS    RESTARTS   AGE
test    1/1     Running   1           1m

$ oc get pod test -o yaml
...
status:
  containerStatuses:
  - name: test
    ready: true
    restartCount: 1
  lastState:
    terminated:
      exitCode: 137
      reason: OOMKilled
  state:
    running:
      phase: Running

```

6.3.5. 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、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、ReplicationController 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
NAME    READY   STATUS    RESTARTS   AGE
test    0/1     Evicted   0           1m

$ oc get pod test -o yaml
...
status:
  message: 'Pod The node was low on resource: [MemoryPressure].'
  phase: Failed
  reason: Evicted

```

6.4. 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の設定

オーバーコミットとは、コンテナ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、容量についての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。



注記

OpenShift Container Platform では、オーバーコミットが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。

6.4.1. 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タ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の制限と要求を調整することができます。



注記

コンテナ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 LimitRanges を注意して設定してください。

6.4.2. リソース要求とオーバーコミットについて

各コンピュータリソースについて、コンテナ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、コンピュータリソースのタイプによって異なります。コンテナが要求または制限を指定しない場合、コンテナ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。

6.4.2.1. Fluentd のバッファチャンクの制限について

Fluentd ロガーが多数のログを処理できない場合、メモリーの使用量を減らし、データ損失を防ぐためにファイルバッファリングに切り換える必要があります。

Fluentd ファイルバッファリングは、記録を **chunks** に保管します。チャンクは **buffers** に保管されません。

Fluentd **buffer_chunk_limit** は、デフォルト値が **8m** の環境変数 **BUFFER_SIZE_LIMIT** によって決定されます。出力ごとのファイルのバッファサイズは、デフォルト値が **256Mi** の環境変数 **FILE_BUFFER_LIMIT** によって決定されます。永続的なボリュームサイズは、**FILE_BUFFER_LIMIT** に出力を乗算した結果よりも大きくなければなりません。

Fluentd Pod では、永続ボリューム **/var/lib/fluentd** は PVC または **hostmount** などによって作成する必要があります。その領域はファイルバッファに使用されます。

buffer_type および **buffer_path** は、以下のように Fluentd 設定ファイルで設定されます。

```
$ egrep "buffer_type|buffer_path" *.conf
output-es-config.conf:
  buffer_type file
  buffer_path `var/lib/fluentd/buffer-output-es-config`
output-es-ops-config.conf:
  buffer_type file
  buffer_path `var/lib/fluentd/buffer-output-es-ops-config`
```

Fluentd **buffer_queue_limit** は変数 **BUFFER_QUEUE_LIMIT** の値です。この値はデフォルトで **32** になります。

環境変数 **BUFFER_QUEUE_LIMIT** は $(\text{FILE_BUFFER_LIMIT} / (\text{number_of_outputs} * \text{BUFFER_SIZE_LIMIT}))$ として計算されます。

BUFFER_QUEUE_LIMIT 変数にデフォルトの値のセットが含まれる場合、以下のようになります。

- **FILE_BUFFER_LIMIT = 256Mi**
- **number_of_outputs = 1**
- **BUFFER_SIZE_LIMIT = 8Mi**

buffer_queue_limit の値は **32** になります。**buffer_queue_limit** を変更するには、**FILE_BUFFER_LIMIT** の値を変更する必要があります。

この数式では、**number_of_outputs** は、すべてのログが単一リソースに送信され、追加のリソースごとに **1** つずつ増分する場合に **1** になります。たとえば、**number_of_outputs** の値は以下のようになります。

- **1:** すべてのログが単一の Elasticsearch Pod に送信される場合
- **2:** アプリケーションログが Elasticsearch Pod に送信され、運用ログが別の Elasticsearch Pod に送信される場合
- **4:** アプリケーションログが Elasticsearch Pod に送信され、運用ログが別の Elasticsearch Pod に送信される場合で、それらがどちらも他の Fluentd インスタンスに転送される場合

6.4.3. コンピュートリソースとコンテナについて

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。

6.4.3.1. コンテナの CPU 要求について

コンテナには要求する CPU の量が保証され、さらにコンテナ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。複数のコンテナが追加の CPU の使用を試行する場合、CPU 時間が各コンテナで要求される CPU の量に基づいて分配されます。

たとえば、あるコンテナが 500m の CPU 時間を要求し、別のコンテナが 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 の比率でコンテナ間で分配されます。コンテナ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。

6.4.3.2. コンテナのメモリー要求について

コンテナには要求するメモリー量が保証されます。コンテナ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。

6.4.4. 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タ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。

各コンピュータリソースについて、コンテナは 3 つの QoS クラスに分類されます (優先順位は降順)。

表6.19 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、コンテナは Burstable として分類されます。
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が最初に強制終了されます。

- **Guaranteed** コンテナは優先順位が最も高いコンテナとして見なされ、保証されます。強制終了されるのは、これらのコンテナで制限を超えるか、またはシステムがメモリ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** コンテナは、制限を超過し、**BestEffort** コンテナ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** コンテナは優先順位の最も低いコンテナとして処理されます。これらのコンテナのプロセスは、システムがメモリ不足になると最初に強制終了されます。

6.4.4.1. Quality of Service (QoS) 層でのメモリ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 QoS 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。

6.4.5. 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 がメモリ制限に達した場合、それぞれのコンテナが 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。

6.4.6. 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。

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
vm.overcommit_memory = 1
```

```
$ sysctl -a |grep panic
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

6.4.6.1. 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。

前提条件

1. 設定したいノードタイプの静的な Machine Config Pool 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. マシンの Config Pool を表示します。

```
$ oc describe machineconfigpool <name>
```

例:

```
$ oc describe machineconfigpool worker

apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  creationTimestamp: 2019-02-08T14:52:39Z
  generation: 1
  labels:
    custom-kubelet: small-pods ❶
```

- ❶ 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、Custom Resource) 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 ❶
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: small-pods ❷
  kubeletConfig:
    cpu-cfs-quota: ❸
      - "false"
```

- ❶ CR に名前を割り当てます。
- ❷ 設定の変更を適用するラベルを指定します。
- ❸ **cpu-cfs-quota** パラメーターを **false** に設定します。

CPU 制限の実施が無効にされる場合、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。

- コンテナが CPU の要求をする場合、これは Linux カーネルの CFS 共有によって引き続き実施されます。

- コンテナがCPUの要求を明示的に指定しないものの、制限を指定する場合には、要求は指定された制限にデフォルトで設定され、Linux カーネルのCFS共有で実施されます。
- コンテナがCPUの要求と制限の両方を指定する場合、要求はLinuxカーネルのCFS共有で実施され、制限はノードに影響を与えません。

6.4.6.2. 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に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 (sshd など)。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。

6.4.7. 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
```

6.4.8. プロジェクトでのオーバーコミットメントを無効化

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
```

6.5. 機能ゲートの使用による各種機能の有効化

管理者は、テクノロジープレビュー機能をオンにすることができます。

6.5.1. 機能ゲートおよびテクノロジープレビュー機能について

機能ゲートのカスタムリソースを使用して、クラスター全体でテクノロジープレビュー機能を有効にすることができます。たとえば、これにより、テクノロジープレビュー機能を実稼働クラスターでは無効にしつつ、これらをテストクラスターで有効にし、完全にテストを実施することができます。



重要

機能ゲートを使用してテクノロジープレビュー機能をオンにした後にそれらをオフにすることはできません。クラスターをアップグレードできなくなります。

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、<https://access.redhat.com/ja/support/offerings/techpreview/> を参照してください。

6.5.2. FeatureGate の影響を受ける機能

以下のテクノロジープレビュー機能が OpenShift Container Platform に含まれています。

FeatureGate	説明	デフォルト
ExperimentalCriticalPodAnnotation	特定の Pod に「Critical」のアノテーションを付け、それらのスケジューリングが保証されるようにします。	True
RotateKubeletServerCertificate	クラスター上でのサーバー TLS 証明書のローテーションを有効にします。	True
SupportPodPidsLimit	Pod でのプロセス数 (PID) を制限するサポートを有効にします。	True
MachineHealthCheck	マシンプール内の正常でないマシンの自動修復を有効にします。	False
CSIBlockVolume	raw ブロックボリュームのサポートを実装するために外部 CSI ドライバーを有効にします。	False
LocalStorageCapacityIsolation	ローカルの一時ストレージの消費、および emptyDir ボリュームの sizeLimit プロパティを有効にします。	False

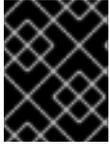
機能ゲートカスタムリソースを編集して、**MachineHealthCheck** および **CSIBlockVolume** 機能を有効にできます。これらの機能をオンにすると元に戻すことができなくなり、クラスターのアップグレードができなくなります。

LocalStorageCapacityIsolation は有効にできません。

6.5.3. 機能ゲートの使用によるテクノロジープレビュー機能の有効化

cluster という名前の機能ゲートカスタムリソースを **openshift-config** プロジェクトで編集し、クラスター内のすべてのノードに対して **MachineHealthCheck** および **CSIBlockVolume** テクノロジープレビュー機能をオンにできます。

有効化



重要

機能ゲートカスタムリソースを使用してテクノロジープレビュー機能をオンに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。

手順

クラスター全体でテクノロジープレビュー機能をオンにするには、以下の手順を実行します。

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 Definitions** ページで、**Actions** → **View Instances** をクリックします。
4. **Feature Gates** ページで、**Create Feature Gates** をクリックします。
5. **featureSet** パラメーターを追加します。

```
apiVersion: config.openshift.io/v1
kind: FeatureGate
metadata:
  name: cluster
spec:
  featureSet: "TechPreviewNoUpgrade" ❶
```

- ❶ **featureSet: "TechPreviewNoUpgrade"** パラメーターを追加します。

6. **Save** をクリックします。