Red Hat OpenShift Pipelines 1.14

パフォーマンスとリソース使用の管理

OpenShift Pipelines でのリソース消費の管理

Last Updated: 2024-03-27

Red Hat OpenShift Pipelines 1.14 パフォーマンスとリソース使用の管理

OpenShift Pipelines でのリソース消費の管理

法律上の通知

概要

このドキュメントでは、OpenShift Pipelines でのリソース消費の管理に関する情報を提供します。

目次

第1章 OPENSHIFT PIPELINES パフォーマンスの管理

OpenShift Pipelines インストールで多数のタスクを同時に実行すると、パフォーマンスが低下する可能性があります。速度の低下やパイプラインの実行の失敗が発生する可能性があります。

参考までに、Amazon Web Services (AWS) **m6a.2xlarge** ノード上で実行されている 3 ノードの OpenShift Container Platform クラスターでの Red Hat テストでは、最大 60 個の単純なテストパイプラインが重大な障害や遅延なく同時に実行されました。より多くのパイプラインを同時に実行すると、失敗したパイプライン実行の数、パイプライン実行の平均期間、Pod 作成のレイテンシー、ワークキューの深さ、および保留中の Pod の数が増加しました。このテストは、Red Hat OpenShift Pipelines バージョン 1.13 で実行されました。バージョン 1.12 から統計的に有意な差は観察されませんでした。

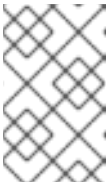
注記

これらの結果は、テスト設定によって異なります。ご使用の設定によるパフォーマンス結果は異なる場合があります。

1.1. OPENSHIFT PIPELINES のパフォーマンスの向上

パイプラインの実行が遅くなったり、失敗が繰り返し発生したりする場合は、次のいずれかの手順を実行して、OpenShift Pipelines のパフォーマンスを向上させることができます。

- OpenShift Pipelines が実行される OpenShift Container Platform クラスター内のノードのリソース使用状況をモニタリングします。リソースの使用率が高い場合は、ノード数を増やします。
- 高可用性モードを有効にします。このモードは、タスク実行とパイプライン実行の Pod を作成して開始するコントローラーに影響します。Red Hat のテストでは、高可用性モードにより、パイプラインの実行時間と、**TaskRun** リソース CR の作成から Pod によるタスク実行の開始までの遅延が大幅に短縮されました。高可用性モードを有効にするには、**TektonConfig** カスタムリソース (CR) で次の変更を加えます。
  - **pipeline.performance.disable-ha** 仕様を **false** に設定します。
  - **pipeline.performance.buckets** 仕様を **5** から **10** までの数値に設定します。
  - **pipeline.performance.replicas** 仕様を **2** より大きく、**pipeline.performance.buckets** 設定以下の数値に設定します。



注記

バケットとレプリカにさまざまな数を試して、パフォーマンスへの影響を確認できます。一般に、数値が大きいほど有益です。CPU やメモリーの使用率など、ノードのリソースの枯渇を監視します。

1.2. 関連情報

- TektonConfig CR を使用したパフォーマンスチューニング

第2章 OPENSHIFT パイプラインのリソース消費の削減

マルチテナント環境でクラスターを使用する場合、各プロジェクトおよび Kubernetes オブジェクトの CPU、メモリー、およびストレージリソースの使用を制御する必要があります。これにより、1 つのアプリケーションがリソースを過剰に消費し、他のアプリケーションに影響を与えるのを防ぐことができます。

結果として作成される Pod に設定される最終的なリソース制限を定義するために、Red Hat OpenShift Pipelines は、それらが実行されるプロジェクトのリソースクォータの制限および制限範囲を使用します。

プロジェクトのリソース消費を制限するには、以下を実行できます。

- リソースクォータを設定し、管理 して、リソースの総消費量を制限します。
- 制限範囲を使用し、リソース消費を制限 します。この対象は、Pod、イメージ、イメージストリームおよび永続ボリューム要求 (PVC) などの特定のオブジェクトのリソース消費です。

2.1. パイプラインでのリソース消費について

各タスクは、**Task** リソースの **steps** フィールドで定義された、特定の順序で実行される多数の必須ステップで設定されます。各タスクは Pod として実行され、各ステップは同じ Pod 内のコンテナーとして実行されます。

ステップは一度に 1 つずつ実行されます。タスクを実行する Pod は、タスク内の 1 つのコンテナーイメージ (ステップ) を一度に実行するのに十分なリソースのみを要求するため、タスク内のすべてのステップのリソースは保存されません。

steps 仕様の **Resources** フィールドは、リソース消費の制限を指定します。デフォルトで、CPU、メモリー、および一時ストレージのリソース要求は、**BestEffort** (ゼロ) 値またはそのプロジェクトの制限範囲で設定される最小値に設定されます。

ステップのリソース要求および制限の設定例

```
spec:
  steps:
  - name: <step_name>
    resources:
      requests:
        memory: 2Gi
        cpu: 600m
      limits:
        memory: 4Gi
        cpu: 900m
```

LimitRange パラメーターおよびコンテナーリソース要求の最小値がパイプラインおよびタスクが実行されるプロジェクトに指定される場合、Red Hat OpenShift Pipelines はプロジェクトのすべての **LimitRange** 値を確認し、ゼロではなく最小値を使用します。

プロジェクトレベルでの制限範囲パラメーターの設定例

```
apiVersion: v1
kind: LimitRange
metadata:
  name: <limit_container_resource>
```

```
spec:
  limits:
  - max:
      cpu: "600m"
      memory: "2Gi"
    min:
      cpu: "200m"
      memory: "100Mi"
    default:
      cpu: "500m"
      memory: "800Mi"
    defaultRequest:
      cpu: "100m"
      memory: "100Mi"
    type: Container
...
```

2.2. パイプラインでの追加のリソース消費を軽減する

Pod 内のコンテナーにリソース制限を設定する場合、OpenShift Container Platform はすべてのコンテナーが同時に実行される際に要求されるリソース制限を合計します。

呼び出されるタスクで一度に 1 つのステップを実行するために必要なリソースの最小量を消費するために、Red Hat OpenShift Pipelines は、最も多くのリソースを必要とするステップで指定される CPU、メモリー、および一時ストレージの最大値を要求します。これにより、すべてのステップのリソース要件が満たされます。最大値以外の要求はゼロに設定されます。

ただしこの動作により、リソースの使用率が必要以上に高くなる可能性があります。リソースクォータを使用する場合、これにより Pod がスケジュールできなくなる可能性があります。

たとえば、スクリプトを使用する 2 つのステップを含むタスクと、リソース制限および要求を定義しないタスクについて考えてみましょう。作成される Pod には 2 つの init コンテナー (エントリーポイントコピー用に 1 つとスクリプトの作成用に 1 つ) と 2 つのコンテナー (各ステップに 1 つ) があります。

OpenShift Container Platform はプロジェクトに設定された制限範囲を使用して、必要なリソース要求および制限を計算します。この例では、プロジェクトに以下の制限範囲を設定します。

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-min-max-demo-lr
spec:
  limits:
  - max:
      memory: 1Gi
    min:
      memory: 500Mi
    type: Container
```

このシナリオでは、各 init コンテナーは要求メモリー 1 Gi (制限範囲の上限) を使用し、各コンテナーは 500 Mi の要求メモリーを使用します。そのため、Pod のメモリー要求の合計は 2 Gi になります。

同じ制限範囲が 10 のステップのタスクで使用される場合、最終的なメモリー要求は 5 Gi になります。これは、各ステップで実際に必要とされるサイズ (500 Mi) よりも大きくなります (それぞれのステップは他のステップの後に実行されるためです)。

そのため、リソースによるリソース消費を減らすには、以下を行います。

- スクリプト機能および同じイメージを使用して、複数の異なるステップを 1 つの大きなステップにグループ化し、特定のタスクのステップ数を減らします。これにより、要求される最小リソースを減らすことができます。
- 相互に独立しており、独立して実行できるステップを、単一のタスクではなく、複数のタスクに分散します。これにより、各タスクのステップ数が減り、各タスクの要求が小さくなるため、スケジューラーはリソースが利用可能になるとそれらを実行できます。

2.3. 関連情報

- OpenShift Pipeline のコンピュートリソースクォータの設定
- プロジェクトごとのリソースクォータ
- 制限範囲によるリソース消費の制限
- リソース要求および制限

第3章 OPENSHIFT PIPELINE のコンピュートリソースクォータの設定

Red Hat OpenShift Pipelines の **ResourceQuota** オブジェクトは、namespace ごとのリソース消費の合計を制御します。これを使用して、オブジェクトのタイプに基づき、namespace で作成されたオブジェクトの数量を制限できます。さらに、コンピュートリソースクォータを指定して、namespace で消費されるコンピュートリソースの合計量を制限できます。

ただし、namespace 全体のクォータを設定するのではなく、パイプライン実行で作成される Pod が使用するコンピュートリソースの量を制限できます。現時点で、Red Hat OpenShift Pipelines ではパイプラインのコンピュートリソースクォータを直接指定できません。

3.1. OPENSHIFT PIPELINE でコンピュートリソース消費を制限する別の方法

パイプラインによるコンピュートリソースの使用量をある程度制御するためには、代わりに、以下のアプローチを検討してください。

- タスクの各ステップでリソース要求および制限を設定します。

  例: タスクのステップごとのリソース要求および制限設定

  ```
  ...
  spec:
    steps:
      - name: step-with-limts
        resources:
          requests:
            memory: 1Gi
            cpu: 500m
          limits:
            memory: 2Gi
            cpu: 800m
  ...
  ```

- **LimitRange** オブジェクトの値を指定して、リソース制限を設定します。**LimitRange** の詳細は、制限範囲によるリソース消費の制限 を参照してください。

- パイプラインのリソース消費を減らします。

- プロジェクトごとにリソースクォータ を設定して管理します。

- 理想的には、パイプラインのコンピュートリソースクォータは、パイプライン実行で同時に実行される Pod が消費するコンピュートリソースの合計量と同じである必要があります。ただし、タスクを実行する Pod はユースケースに基づきコンピュートリソースを消費します。たとえば、Maven ビルドタスクには、ビルドするアプリケーションごとに異なるコンピュートリソースが必要となる場合があります。その結果、一般的なパイプラインでタスクのコンピュートリソースクォータを事前に定義できません。コンピュートリソースの使用に関する予測可能性や制御性を高めるには、さまざまなアプリケーション用にカスタマイズされたパイプラインを使用します。

注記

ResourceQuota オブジェクトで設定される namespace で Red Hat OpenShift Pipelines を使用する場合、タスク実行およびパイプライン実行がエラーで失敗する可能性があります (例: **failed quota: <quota name> must specify cpu, memory**)。

このエラーを回避するには、以下のいずれかを実行します。

- (推奨) namespace の制限範囲を指定します。
- すべてのコンテナーの要求および制限を明示的に定義します。

詳細は、問題 および 解決策 を参照してください。

これらの方法で対応できないユースケースには、優先順位クラスのリソースクォータを使用して回避策を実装できます。

3.2. 優先順位クラスを使用したパイプラインリソースクォータの指定

PriorityClass オブジェクトは、優先順位クラス名を、相対的な優先順位を示す整数値にマッピングします。値が大きいと、クラスの優先度が高くなります。優先順位クラスの作成後に、仕様に優先順位クラス名を指定する Pod を作成できます。さらに、Pod の優先順位に基づいて、Pod によるシステムリソースの消費を制御できます。

パイプラインにリソースクォータを指定することは、パイプライン実行が作成する Pod のサブセットのリソースクォータを設定することに似ています。以下の手順では、優先順位クラスに基づいてリソースクォータを指定して回避策の例を提供します。

手順

1. パイプラインの優先順位クラスを作成します。

   例: パイプラインの優先順位クラス

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: pipeline1-pc
value: 1000000
description: "Priority class for pipeline1"
```

2. パイプラインのリソースクォータを作成します。

   例: パイプラインのリソースクォータ

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pipeline1-rq
spec:
  hard:
    cpu: "1000"
    memory: 200Gi
    pods: "10"
  scopeSelector:
```

```
    matchExpressions:
    - operator : In
      scopeName: PriorityClass
      values: ["pipeline1-pc"]
```

3. パイプラインのリソースクォータの使用量を確認します。

   例: パイプラインにおけるリソースクォータ使用状況の確認

```
$ oc describe quota
```

   出力例

```
Name:       pipeline1-rq
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     1k
memory      0     200Gi
pods        0     10
```

   Pod が実行されていないため、クォータは使用されません。

4. パイプラインおよびタスクを作成します。

   例: パイプラインの YAML

```
apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: maven-build
spec:
  params:
  - name: GIT_URL
  workspaces:
  - name: local-maven-repo
  - name: source
  tasks:
  - name: git-clone
    taskRef:
      kind: ClusterTask
      name: git-clone
    params:
      - name: url
        value: $(params.GIT_URL)
    workspaces:
    - name: output
      workspace: source
  - name: build
    taskRef:
      name: mvn
    runAfter: ["git-clone"]
    params:
    - name: GOALS
```

```
      value: ["package"]
    workspaces:
    - name: maven-repo
      workspace: local-maven-repo
    - name: source
      workspace: source
  - name: int-test
    taskRef:
      name: mvn
    runAfter: ["build"]
    params:
    - name: GOALS
      value: ["verify"]
    workspaces:
    - name: maven-repo
      workspace: local-maven-repo
    - name: source
      workspace: source
  - name: gen-report
    taskRef:
      name: mvn
    runAfter: ["build"]
    params:
    - name: GOALS
      value: ["site"]
    workspaces:
    - name: maven-repo
      workspace: local-maven-repo
    - name: source
      workspace: source
```

例: パイプラインのタスクの YAML

```
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: mvn
spec:
  workspaces:
  - name: maven-repo
  - name: source
  params:
  - name: GOALS
    description: The Maven goals to run
    type: array
    default: ["package"]
  steps:
    - name: mvn
      image: gcr.io/cloud-builders/mvn
      workingDir: $(workspaces.source.path)
      command: ["/usr/bin/mvn"]
      args:
        - -Dmaven.repo.local=$(workspaces.maven-repo.path)
        - "$(params.GOALS)"
```

5. パイプライン実行を作成して開始します。

例: パイプライン実行の YAML

```
apiVersion: tekton.dev/v1
kind: PipelineRun
metadata:
  generateName: petclinic-run-
spec:
  pipelineRef:
    name: maven-build
  params:
  - name: GIT_URL
    value: https://github.com/spring-projects/spring-petclinic
  taskRunTemplate:
    podTemplate:
      priorityClassName: pipeline1-pc
  workspaces:
  - name: local-maven-repo
    emptyDir: {}
  - name: source
    volumeClaimTemplate:
      spec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 200M
```

注記

パイプライン実行は、エラー: **failed quota: <quota name> must specify cpu, memory** で失敗する可能性があります。

このエラーを回避するには、namespace の制限範囲を設定します。ここで、ビルドプロセス中に作成された Pod に **LimitRange** オブジェクトのデフォルトが適用されます。

制限範囲の設定の詳細は、**関連情報** セクションの **制限範囲によるリソース消費の制限** を参照してください。

6. Pod の作成後に、パイプライン実行におけるリソースクォータの使用状況を確認します。

例: パイプラインにおけるリソースクォータ使用状況の確認

```
$ oc describe quota
```

出力例

```
Name:       pipeline1-rq
Namespace:  default
Resource    Used  Hard
--------    ----  ----
```

```
cpu         500m  1k
memory      10Gi  200Gi
pods        1     10
```

この出力は、優先クラスごとにリソースクォータを指定することで、特定の優先クラスに属するすべての同時実行 Pod のリソースクォータをまとめて管理できることを示しています。

3.3. 関連情報

- 制限範囲によるリソース消費の制限
- Kubernetes のリソースクォータ
- Kubernetes の制限範囲
- リソース要求および制限