New Relic.

Kubernetesの基礎知識

クラスタ容量からコンテナのヘルスチェック、ストレージボリュームの管理まで、5つのKubernetesの基礎知識を身に付けましょう。

目次

序文	01
第1章：リクエストとリミットを使ったクラスタ容量の管理方法	02
第2章：ヘルスチェックの使い方	06
第3章：Kubernetes Secrets の使い方	11
第4章：クラスタの整理	18
第5章：Kubernetes ボリュームの操作	25

序文

Cloud Native Computing Foundation(CNCF)が実施した調査によると、回答者の83%がKubernetesを本番環境で使用しており、2018年の58%から増加傾向にあることが明らかになりました。コンテナ化されたアプリケーションのデプロイ、スケーリング、管理の自動化に関しては、Kubernetesを使用するメリットをさまざまなチームが認めています。ただし、このテクノロジーを理解するのは容易なことではありません。Kubernetesの導入を進める中で、プラットフォームのあまりの範囲の広さに圧倒されるでしょう。

本書では、クラスタ容量の管理、ヘルスチェックの使用、クラスタの整理など、Kubernetesの基礎知識を5つの章に分けて、解説しています。Kubernetesの初心者には必須の情報ですが、プロの場合でも、Kubernetesの管理を成功させるための基本的な要素を確認できると思います。

第1章:リクエストとリミットを使ったクラスタ容量の管理方法

Kubernetesはクラスタ内のノードを管理しますが、まずアプリケーションのリソース要件を定義する必要があります。コンテナを円滑に稼働させるには、Kubernetesのリソースの管理方法、特にピーク時の管理方法を把握することが重要です。

本章では、リクエストとリミットを使用して、KubernetesがCPUとメモリを管理する方法について説明します。

リクエストとリミットの仕組み

Kubernetesクラスタの各ノードには、コンテナの実行に使用できるメモリ(RAM)と演算能力(CPU)が割り当てられています。

Kubernetesでは、1つ以上のコンテナを論理的にグループ化した「Pod」を定義しています。Podは、ノードの上にデプロイされ、管理されます。Podを作成する際は、Pod内でコンテナが共有するストレージとネットワークを指定します。次に、Kubernetesのスケジューラは、Podの実行に必要なリソースを持つノードを探します。

スケジューラを支援するため、リクエストとリミットを使用して、各コンテナのRAMとCPUの上限と下限を指定します。この2つのキーワードで、次の項目を指定できます。

- コンテナにリクエストを指定することで、コンテナに必要なRAMやCPUの最小値を設定します。KubernetesがPodリクエストの合計にコンテナリクエストを取り込みます。スケジューラは、この合計リクエストを使用して、十分なリソースを持つノードにPodをデプロイできるようにします。
- コンテナにリミットを指定することで、コンテナに必要なRAMやCPUの最大値を設定します。Kubernetesは、リミットを実行するコンテナサービス(たとえば、Docker)にリミットを変換します。コンテナがメモリのリミットを超えた場合、可能であれば終了させ、再起動させます。CPUの制限はそれほど厳密ではないため、長期間、超過しても問題ありません。

リクエストとリミットの使い分けを見てみましょう。

CPUのリクエストとリミット設定

CPUのリクエストとリミットをCPU単位で測定します。Kubernetesでは、シングルCPUユニットは、クラウドプロバイダーでは、仮想CPU(vCPU)またはコア、ベアメタルプロセッサーではシングルスレッドに相当します。

状況によっては、1台のフルCPUユニットは、コンテナに対するリソースとしてみなされます。特に、マイクロサービスにおいては、いまだにそうです。KubernetesがCPUの端数(fraction)をサポートしているのはこのためです。

たとえば「CPUの0.5」など、CPUの端数は小数点で入力できますが、Kubernetesでは「millicpu」表記を使用しています。1,000millicpu(または1,000m)が1CPUに相当します。

New Relic.

CPUユニット、またはその一部のリクエストを送信すると、Kubernetesスケジューラがこの値を使用して、Podが実行できるクラスタ内のノードを見つけます。たとえば、Podに1CPUのCPUリクエストを持つコンテナが1つ含まれている場合、スケジューラは、このPodを配置するノードに1つのCPUリソースがあることを確認します。Dockerコンテナの場合、Kubernetesは、CPU共有制限を使用してCPUを比例させます。

リミットを指定すると、KubernetesはコンテナのCPU使用率の上限を設定しようとします。前述したように、これはハードリミットではなく、コンテナ化テクノロジーによっては、このリミットを超える場合も、超えない場合もあります。Dockerコンテナの場合、KubernetesはCPU周期制限を使用してCPU使用率の上限を設定します。これにより、Dockerはコンテナが使用する100ミリ秒以上の実行時間の割合を制限することができます。

CPUリクエストが0.5ユニット、CPUリミットが1.5ユニットのPod構成YAMLファイルのサンプルを以下に紹介します。

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-request-limit-example
spec:
  containers:
  - name: cpu-request-limit-container
    image: images.example/app-image
    resources:
      requests:
        cpu: "500m"
      limits:
        cpu: "1500m"
```

この構成では、リソースのセクションで指定されたイメージリミットのある「cpu-request-limit-container」というシングルコンテナを定義します。このセクションでは、リクエストとリミットを指定します。この場合、500millicpu（0.5またはCPUユニットの50%）を要求し、コンテナを1500millicpu（1.5またはCPUユニットの150%）に制限することになります。

メモリのリクエストとリミット設定

メモリのリクエストとリミットは、「バイト」で測定されますが、(K)キロバイト（1,000バイト）、メガバイト（M）、ギガバイト（G）、1,000,000,000バイトなど、大きな値を指定する標準ショートコードもあります。また、これらのショートカットには、2のべき乗に相当するものもあります。たとえば、Ki（1,024バイト）、Mi、Giなどです。CPUユニットとは違い、メモリは最小単位が1バイトなので端数はありません。

Kubernetesスケジューラは、メモリのリクエストを使用して、クラスタ内でPodを実行するのに十分なメモリを持つノードを探します。メモリのリミットは、CPUリミットと同様に動作しますが、厳密に実施されます。コンテナがメモリのリミットを超えた場合、コンテナは終了し、「out of memory（メモリ不足）」エラーで再起動される可能性があります。

メモリのリクエストが256メガバイト、メモリのリミットが512メガバイトのPod構成YAMLファイルのサンプルを以下に紹介します。

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-request-limit-example
spec:
  containers:
  - name: memory-request-limit-container
    image: images.example/app-image
    resources:
      requests:
        memory: "256M"
      limits:
        memory: "512M"
```

この構成では、リソースのセクションで指定されたイメージリミットのある「memory-request-limit-container」というシングルコンテナを定義します。メモリのリクエストを256Mに指定し、コンテナを512Mに制限しています。

名前空間(namespaces)でリミットを設定

複数の開発者、または開発者のチームが共通のKubernetesクラスタで作業している場合、リソースの浪費を防ぐには、共通のリソース要件を設定しておくことをおすすめします。Kubernetesでは、チームごとに定義した名前空間(namespaces)に、リソースクォータを割り当てることができます。

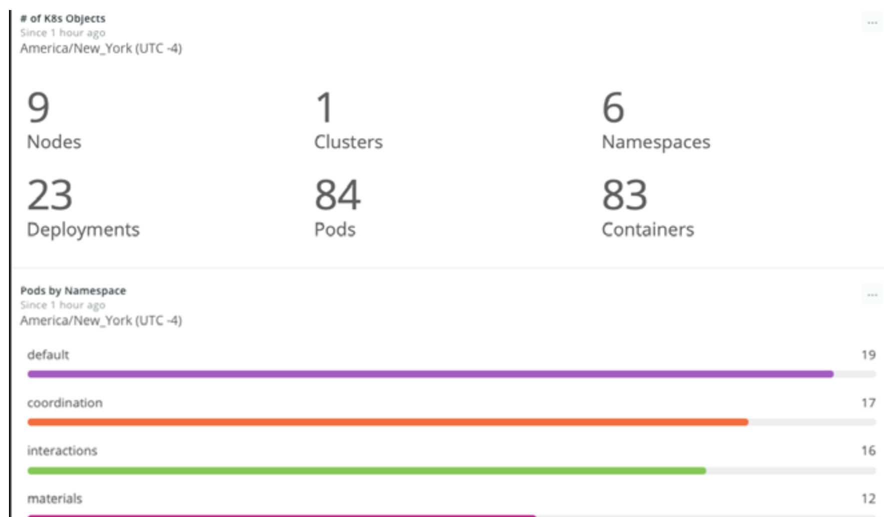
たとえば、64CPUユニットと256ギガバイトのRAMを持つKubernetesクラスタを8つのノードに分散して、配置することができます。CPUユニットが10、メモリが80ギガバイトのリソースクォータで、開発チームごとに3つの名前空間を作成できます。これにより、開発チームはCPUとメモリに余裕を持たせたまま、希望する数のPod(上限まで)を作成できます。

名前空間のリソース割り当てを指定する方法については、Kubernetes資料の「Resource Quotas(リソースクォータ)」セクションをご覧ください。

クラスタ容量の監視の重要性

コンテナと名前空間の両方にリクエストとリミットを設定すれば、リソース不足の予防策として、大きな効果が得られます。それぞれのサービスの健全性だけでなく、クラスタ全体の健全性を維持する上で、重要な役割を果たすのが監視(モニタリング)です。

Kubernetes Pod内で複数のサービスが実行する大規模クラスタの場合、ヘルスチェックやエラー監視が複雑になります。New Relicなどの可観測性ツールを使用すれば、Kubernetesクラスタや実行するサービスが監視しやすくなります。こうすることで、コンテナやクラスタ全体に適切なリクエストやリミットを設定できるようになります。



Kubernetesクラスタに常に十分な容量を確保するには、KubernetesのCPUリソースとメモリリソースの取り扱い方を理解し、管理できるように構成することが重要です。ここで説明したように、CPUとメモリのリクエストやリミットの設定は非常に簡単です。監視レイヤーを強化することで、Podがクラスタのリソースに対して競合しないように設定できます。

第2章：ヘルスチェックの使い方

Kubernetesで効率的にコンテナを管理するには、コンテナが正常に動作し、トラフィックを受信していることを確認するため、コンテナの状態をチェックする必要があります。Kubernetesでは、ヘルスチェック（プローブとも呼ばれる）を使って、アプリのインスタンスが稼働しているか、適切に応答するかを確認します。

本章では、さまざまなプローブの種類と使い方を説明します。

プローブが重要な理由

分散されたシステムは、管理しにくいという問題があります。各コンポーネントは独立して動作するので、他のコンポーネントが故障しても、継続して動作します。アプリケーションは、何らかのタイミングでクラッシュするかもしれませんし、あるいは、初期化段階で、リクエストを受信して処理する準備ができていない可能性もあります。

すべてのコンポーネントが正常に動作していなければ、システムの状態を判断できません。プローブを使うことで、コンテナの状態（稼働/停止）をチェックし、他のコンテナからのKubernetesへのアクセスを一時的に停止するべきかどうかを判断できます。Kubernetesは、Pod全体の健全性を判断するため、各コンテナの状態を検証します。

プローブの種類

分散型アプリケーションを導入・運用する際は、コンテナを作成し、起動し、実行し、終了します。Kubernetesでは、さまざまなタイプのプローブを使用して、コンテナのライフサイクルの各段階における状態をチェックしています。

Livenessプローブは、アプリが正常に稼働しているかどうかをチェックします。各ノードで実行されるKubeletエージェントは、Livenessプローブを使用して、コンテナが正常に動作していることを確認します。コンテナアプリがリクエストに対応しなくなった場合、Kubeletが介入し、コンテナを再起動します。

たとえば、アプリケーションがデッドロック（ロック解除待ちの状態）で応答せず、処理が停止してしまった場合、Livenessプローブがアプリケーションに不具合があることを検出します。Kubeletは、このような状態のコンテナを終了し、再起動します。アプリケーションにさらにデッドロックを引き起こす不具合（バグ）がある場合でも、再起動によってコンテナの可用性が向上します。これによって、開発者が不具合を特定し、解決する時間を確保することもできます。

コンテナのライフサイクル全体で実行するのが、**Readinessプローブ**です。Kubeletは、このプローブを使用して、コンテナがトラフィックを受け入れられる状態であるかを認識します。Readinessプローブが失敗する場合、プローブが成功するまで、KubernetesはPodへのトラフィックの送信を停止します。

たとえば、コンテナが、ファイルの解凍、インデックス作成、データベーステーブルの投入など、初期化タスクを実行する必要があるとします。起動プロセスが

完了するまで、コンテナはトラフィックを受信したり、サービスを提供することができません。この場合、Readinessプローブは失敗し、Kubernetesは他のコンテナにリクエストをルーティングします。

すべてのコンテナの準備が整った時点で、Podも準備できているとみなされます。これにより、KubernetesはどのPodをサービスのバックエンドとして使用するかをコントロールすることができます。準備ができていない場合、Podはサービスのロードバランシングから切り離されます。

Startupプローブは、コンテナアプリケーションが正常に初期化されたかどうかを判断するために使用されます。Startupプローブが失敗すると、Podが再起動されます。

Podコンテナの準備に時間がかかりすぎると、Readinessプローブが繰り返し失敗する可能性があります。この場合、コンテナが起動する前にkubeletによって終了させられる危険性があります。そこで、役立つのがStartupプローブです。

Startupプローブが成功するまでは、アプリケーションの起動に干渉しないように、LivenessプローブとReadinessプローブによるチェックを強制的に無効にします。起動に時間がかかるレガシーアプリケーションの場合、特に便利です。

プローブの作成

ヘルスチェックプローブを作成する場合、コンテナに対してリクエストを発行する必要があります。Kubernetes Liveness、Readiness、Startupプローブを実装する場合、3種類の方法があります。

1. HTTPリクエストを送信する
2. コマンドを実行する
3. TCPソケットを開く

HTTPリクエスト

HTTPリクエストは、一般的で、わかりやすいLivenessプローブの作成方法のひとつです。HTTPエンドポイントを公開するには、コンテナ内に任意の軽量なHTTPサーバーを実装します。

Kubernetesプローブは、コンテナのIPのエンドポイントに対してHTTP GETリクエストを実行し、サーバーが稼働しているかどうかをチェックします。エンドポイントが成功コードを返した場合、コンテナは正常に稼働しているとみなされます。それ以外の場合は、kubeletは、コンテナを終了し、再起動します。

たとえば、「k8s.gcr.io/liveness」というイメージに基づいたコンテナがあるとします。この場合、HTTP GETリクエストを使用するLivenessプローブを定義すると、YAML構成ファイルは次のようなスニペットになります。

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

```
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

この構成では、initialDelay-SecondsおよびperiodSecondsプロパティを持つシングルコンテナPodを定義し、3秒ごとにLivenessプローブを実行し、3秒待ってから最初のプローブ実行するよう、kubeletを設定しています。Kubeletは、`8080`ポートで/healthzパスにリクエストを送信することで、コンテナが正常に稼働していることをチェックし、成功の結果コードを受けます。

コマンド

HTTPリクエストが使用できない場合は、コマンドプローブを使用します。

コマンドプローブを設定すると、kubeletはターゲットコンテナで`cat/tmp/healthy`コマンドを実行します。コマンドが成功すれば、コンテナが正常に稼働しているとみなされます。それ以外の場合は、Kubernetesはコンテナを終了し、再起動します。

「k8s.gcr.io/busybox」のイメージに基づいたコンテナを実行するPodのYAML構成は、以下のようになります。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/
healthy; sleep 600
```

```
livenessProbe:
  exec:
    command:
    - cat
    - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

この構成では、`initialDelaySeconds`およびperiodSecondsキーを持つシングルコンテナPodを定義し、5秒ごとにLivenessプローブを実行し、最初のプローブ完了まで5秒待機するよう、kubeletを設定しています。

Kubeletは、コンテナで`cat/tmp/healthy`コマンドを実行し、プローブを起動します。

TCP接続

TCPソケットプローブが定義されている場合、Kubernetesはコンテナの指定したポートでTCP接続を開きます。Kubernetesが成功した場合、コンテナは正常に稼働しているとみなされます。HTTPプローブやコマンドプローブでは不十分な場合、TCPプローブが役立ちます。コンテナでTCPプローブが有効なケースとしては、TCPプロトコルのインフラが既に存在するgRPCやFTPサービスなどが挙げられます。

以下の構成で、kubeletは指定されたポートでコンテナへのソケットを開きます。

```
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: k8s.gcr.io/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

上記の構成は、HTTPチェックのケースと同様です。ReadinessとLivenessプローブを定義します。コンテナが起動すると、kubeletは5秒間待機して最初のReadinessプローブを送信します。その後、kubeletは10秒ごとにコンテナの準備状況を確認します。

Kubernetesの状態のモニタリング

プローブは、コンテナが正常かどうかを示すのみで、それ以外の情報は通知しません。

複数のノードにデプロイされているKubernetes Podでサービスを実行している場合、ヘルスチェックやエラー監視は複雑になります。

Kubernetesプラットフォームを扱う開発者やDevOpsスペシャリストなら、New RelicがKubernetesのヘルスチェックを実行し、インサイトを収集、コンテナの問題をトラブルシューティングする優れたツールであることはお気づきでしょう。

クラスタ内のコンテナに障害がないことを確認する際、プローブによるヘルスチェックが不可欠です。KubernetesはLiveness、Readiness、Startupプローブを使用して、コンテナの再起動が必要かどうか、あるいはPodをサービスから削除する必要があるかどうかを判断します。こうしたチェック機構を実行することで、分散システムのサービスの信頼性と可用性を確保できます。

第3章：Kubernetes Secretsの使い方

Kubernetesで動作するコンテナ型アプリケーションは、外部リソースへのアクセスが必要な場合が多く、ほとんどがシークレット、パスワード、キー、トークンを使用します。Kubernetes Secretsには、これらのアイテムを安全に保管できるので、Pod定義やコンテナイメージに保管する必要はありません。

本章は、Kubernetesでシークレットを作成・使用するさまざまな方法を紹介します。環境に最適なアプローチを選択する際の参考にしてください。

Kubernetes Secretsを作成する

Kubernetes Secretsでシークレットを作成し、保管する方法は、以下の3通りです。

- コマンドラインを使用する
- 構成ファイルを使用する
- ジェネレータを使用する

上記の方法で、シークレットを作成するプロセスを紹介します。

コマンドラインからKubernetes Secretsを作成する

Kubernetes管理者用コマンドラインツール、`kubectl`を使用してシークレットを作成できます。このツールでは、ファイルを使用する、ローカルコンピューターからリテラル文字列を渡す、シークレットにパッケージ化する、APIを使用してクラスタサーバーでオブジェクトを作成するなどの操作が可能です。シークレットオブジェクトは、DNSサブドメイン名の形式であることに注意してください。

ユーザー名とパスワードのシークレットは、以下のコマンドラインのパターンを使用します。

```
kubectl create secret generic <secret-object-name>
<flags>
```

公開鍵/秘密鍵のペアからTLS（Transport Layer Security）を使用したシークレット作成の場合は、以下のコマンドラインパターンを使用します。

```
kubectl create secret tls <secret-object-name>
--cert=<cert-path> --key=<key-file-path>
```

また、データベース用のユーザー名とパスワードの組み合わせを使って、一般的なシークレットを作成することもできます。この例では、リテラルフラグを適用し、コマンドプロンプトでユーザー名とパスワードを指定しています。

```
kubectl create secret generic sample
-db-secret --from-literal=username=admin
--from-literal=password='7f3,F9D^LJz37]!W'
```

このコマンドは、ユーザー名をadmin、パスワードを7f3,F9D^L-Jz37]!Wとして、sample-db-secretという新しいシークレットを作成します。ここで注意したいのは、強力で複雑なパスワードは、特殊文字がエスケープされるケースが多いということです。これを避けるには、すべてのユーザー名とパスワードをテキストファイルに入れ、以下のフラグを使用します。

```
kubectl create secret generic sample-db-secret --from-
file=username.txt --from-file=password.txt
```

このコマンドは、この情報を含むファイル名を提供するので、ユーザー名とパスワードキーをドロップします。キーがファイル名と異なる場合は、--from-literalスイッチと同じ方法で、--from-literalスイッチに戻すことができます。

```
kubectl create secret generic sample-db-secret
--from-file=username=123.txt --from-file=password=xyz.txt
```

構成ファイルでKubernetes Secretsを設定する

JSONやYAML構成ファイルを使って、シークレットを作成する方法もあります。構成ファイルで作成したシークレットのデータマップは、dataおよびstringDataの2種類です。前者は、値がbase64でエンコードされている必要がありますが、後者はエンコードされていない文字列として、値を提供することができます。

YAMLファイルのシークレットには、以下のテンプレートを使用します。

```
apiVersion: v1
kind: Secret
metadata:
    name: <secret name>
type: Opaque
data:
    <key>: <base64 Value>
stringData:
    <key>: <string value>
```

kubectl apply -f ./<file- name>.yamlコマンドを使用して、テンプレートを適用します。例として、複数のシークレットの値が必要なアプリケーションのYAMLファイルを示します。

```
apiVersion: v1
kind: Secret
metadata:
```


New Relic.

```
    name: my-example-app
type: Opaque
data:
    app-user: YWRtaW5pc3RyYXRvcg==
    app-password: cGFzc3dvcmQ=
stringData:
    Dbconnection:
Server=tcp:myserver.database.
net,1433;Database=myDB;User ID=mylogin@myserver;Passwor
d=myPassword;Trusted_Connection=False;Encrypt=True;
    config.yaml: |-
        LogLevel: Warning
        API_TOKEN: NcNIMcMYMAMg.MGwjPnPfEBgqMl8Q
        API_URI: https://www.myapp.com/api
```

上記のYAMLファイルには、以下の値が含まれています。

- The secret name (my-example-app)
- An `app-user` (“administrator,” base64-encoded)
- An `app-pasword` (“password,” base64-encoded)
- A `dbconnection` string
- The `config.yaml` file with data

複数のシークレットや機密性の高い設定情報を単一の構成ファイルにパッケージ化する際に適した方法です。

ジェネレータでKubernetes Secretsを作成する

シークレット作成の3つ目は、「`kustomization.yaml`」という構成ファイルを使用してKubernetesオブジェクトをカスタマイズするスタンドアロンツール、Kustomizeを使用する方法です。

Kustomizeでは、ファイル(各行にキーと値のペアを記述)または構成ファイル内のリテラルでシークレットを指定することにより、コマンドラインと同様の方法でシークレットを生成することができます。

シークレットについては、「kustomiza-tion.yaml」ファイル内で、以下の構造を使用します。

```
secretGenerator:
    name: <secret-name>
    files:
        <filename>
    literals:
        <key>=<value>
```

「kustomization.yaml」ファイルを作成し、ディレクトリ内のすべてのリンクファイルを含めると、kubectl kustom- ize <directory>コマンドを使用した後、kubectl apply -k <directory>コマンドで設定を適用します。

次の例の「`kustomization.yaml`」ファイルでは、2つのリテラルキー/値(API_TOKENとAPI_URI)を持つsecretとconfig.yamlファイルを作成しています。

```
secretGenerator:
    name: example-app-secrets
    files:
        passwords.txt
    literals:
        API_TOKEN: NcNIMcMYMAMg.MGwjPnPfEBgqMl8Q
        API_URI: https://www.myapp.com/api
```

この例で引用した「`config.yaml`」ファイルは、アプリケーションの構成ファイルである可能性があります。

最適なKubernetes Secretsの作成方法はどれですか?

ここで挙げた方法は、特定の状況では「最適」です。

Podに追加する1つまたは2つのシークレット(たとえば、ユーザー名とパスワード)があり、それがローカルファイルにある場合、またはリテラルとして渡す場合は、コマンドラインが最適です。

構成ファイルは、Podに含める少数のシークレットを一括して処理する場合に最適です。

Kustomize構成ファイルは、複数の構成ファイル、複数のPodにデプロイしたいシークレットがある場合、推奨されます。

シークレットを作成したら、主に2つの方法でアクセスできます。

- ファイルを使用する(ボリュームマウント型)
- 環境変数を使用する

前者の方法は、アプリケーションのプロセスの一環として、構成ファイルにアクセスする方法に似ています。後者の方法は、アプリケーションがアクセスするための環境変数としてシークレットをロードします。ここでは、この二つの方法を紹介します。

ボリュームマウント型Kubernetes シークレットにアクセスする

ボリュームにロードされたシークレットにアクセスするには、spec[].[] `volumes[].secret.secret- Name`でPodにシークレットを追加する必要があります。次に、`spec[]. containers[].volumeMounts`で各コンテナにボリュームを追加します。この場合、ボリュームの名前はシークレットの名前と同じで、readOnly(読み取り専用)が`"true"`に設定されています。

また、追加で指定できるオプションも多数あります。単一コンテナのPodの例を見てみましょう。

```
apiVersion: v1
kind: Pod
metadata:
    name: mypod
spec:
    containers:
```

```
        - name: myapp
          image: ubuntu
          volumeMounts:
            - name: secrets
              mountPath: "/etc/secrets"
              readOnly: true
volumes:
    - name: secrets
      secret:
          secretName: mysecret
          defaultMode: 0400
          items:
            - key: username
              path: my-username
```

この構成ファイルは、シングルコンテナ(myapp)のシングルPod(mypod)を指定します。Podのボリュームセクションにあるsecretsというボリュームは、すべてのコンテナで共有されています。このボリュームはsecretタイプで、mysecretというシークレットをロードします。UNIXファイル権限0400を使用してボリュームをロードします。これにより、所有者(root)に読み取りアクセス権が与えられますが、他のユーザーからはアクセスできません。

シークレットには含まれるアイテムリストは、特定の秘密キー(この場合は、username)と追加されたパス(my-username)のみをキャストします。コンテナ(myapp)で、volumeMounts(secrets)のボリュームをmountPathに読み取り専用でマッピングします。

ユーザー名をmy-usernameにキャストしているため、コンテナのディレクトリ/etc/secretsは次のようになります。

```
lrwxrwxrwx 1 root root 2 September 20 19:18 my-username
-> ..data/username
```

1つの値をキャストしてパスを変更しているため、キーが変更されています。symlinkを追ってみると、権限が正しく設定されていることがわかります。

```
-r-------- 1 root root 2 September 20 19:18 username
```

シークレットボリュームのすべてのファイルには、シークレットのbase64-decoded値が含まれます。

シークレットを更新または修正すると、ボリュームも更新されるので、アプリケーションがシークレットを再読み取りできるようになるのが、ボリュームにシークレットをロードする方法の利点です。また、シークレットファイル(機密性の高い構成ファイルなど)の解析や複数のシークレットの参照も簡単になります。

環境変数でKubernetes Secretsにアクセスする

環境変数を使って、コンテナにシークレットをプロジェクション(投影)する方法もあります。この方法では、env[].valueFrom.secretKey Refを使って、追加したいシークレットキーごとに環境変数を追加します。Podの仕様の例を見てみましょう。

```
apiVersion: v1
kind: Pod
metadata:
    name: mypod
spec:
    containers:
        - name: myapp
        image: ubuntu
        env:
          - name: USERNAME
          valueFrom:
              secretKeyRef:
                  name: mysecret
                  key: username
          - name: PASSWORD
          valueFrom:
              secretKeyRef:
                  name: mysecret
                  key: password
```

この構成ファイルでは、mysecret secretの2つのキー(`username`と`password`)を環境変数としてコンテナに渡します。これらの値は、シークレットをbase64でデコーディング(復号)した値です。

コンテナにログインして、echo `$USERNAME`コマンドを実行すると、`username`シークレットのデコーディング値("admin"など)が表示されます。

この方法でシークレットをキャストする最大の利点は、シークレットの値を具体的に設定できることです。さらに、アプリケーションによっては、環境変数を読み込む方が、構成ファイルを解析するよりも簡単です。

Kubernetes Secretsの代替策

Kubernetesクラスタでのシークレット管理は比較的シンプルかつ安全で、ほとんどの要件を満たしますが、欠点もあります。特に、シークレットは「Pod」などの名前空間を使用するため、シークレットとPodの名前空間が同一の場合、すべてのPodがシークレットを読み取れることになります。

さらに、キーが自動的に回転しないため、手動でローテーションさせなければならないことも欠点の一つです。

こうした問題を解消し、集中的なシークレット管理を行うには、以下のような代替構成を使用することもできます。

- Hashicorp VaultやAWS Secrets Managerなどのクラウドベンダーシークレット管理ツールを統合する。これらのツールは通常、Kubernetesサービスアカウントを使用して、シークレットのコンテナへのアクセスを許可し、webhookを変更してシークレットをPodにマウントします。
- AWS Identity and Access Managerなど、クラウドベンダーのIAM(Identity and Access Management)ツールを統合する。この種類の統合は、WebアプリケーションのOpenID Connectと同様の方式で、KubernetesがSecure Token Serviceのトークンを利用できるようにするものです。

- PodにサイドカーとしてロードされたConjurなどのサードパーティのシークレットマネージャを実行する。

Kubernetesでコンテナを実行する場合、ほぼすべてのアプリケーションがデータベースやサービスなどの外部リソースにアクセスする必要があるため、シークレットの安全な保管は重要な要素です。Kubernetes Secretsを使用することで、クラスタ全体の機密性の高いアプリケーション情報を管理でき、非一元的な方法でシークレットを管理するリスクを最小限に抑えることができます。

第4章：クラスタの整理

Kubernetesはスケーリング（拡張性）を想定して設計されています。そのため、小規模なクラスタから始めて、徐々に基盤を拡大することもできます。しばらく経てば、同じクラスタが数十台のPodや数百台のコンテナ（あるいはそれ以上）を実行している可能性もあります。

ただし、整理しておかなければ、デプロイしたサービスやオブジェクトの数が無秩序に増えて制御できなくなり、パフォーマンスやセキュリティの問題につながる恐れがあります。

本章では、Kubernetesクラスタの整理に便利な3種類のツール（名前空間、ラベル、アノテーション（注釈））について説明します。

名前空間の仕組み

デフォルトでは、Kubernetesは物理的なクラスタ（すべてのKubernetesオブジェクトを作成）に実行可能な名前空間を1つ作成します。しかし、プロジェクトの規模が拡大すると、最終的には1つの名前空間では限界が来る可能性があります。

ここで重要なのは、名前空間は仮想クラスタとして考えることができ、Kubernetesは複数の仮想クラスタをサポートするということです。複数の名前空間を設定することで、高度なファサードが完成します。

チームは単一の名前空間での作業から解放され、管理性、セキュリティ、パフォーマンスが向上します。

各企業のチーム規模、構造、プロジェクトの複雑さなどの要因に応じて、名前空間戦略は異なります。少数のマイクロサービスを行う小規模のチームなら、すべてのサービスをデフォルトの名前空間に簡単にデプロイできます。しかし、急成長中の企業の場合はサービスの数も多く、単一の名前空間では、チームの作業調整にも限界が生じます。この場合、企業がサブチームを作り、それぞれに別の名前空間を設けることができます。

また、大企業では、チームが広範囲に分散し、お互いにプロジェクトの内容を知らないことも多く、変更の頻度が高いと、対応するのが難しくなります。サードパーティ企業もプラットフォームに関わり、複雑度も高くなります。調整するリソースの数が多すぎると、管理上の問題につながり、開発者にとっても、ローカルマシンでスタック全体を実行するのは不可能です。サービスメッシュやマルチクラウドの継続的デリバリなどのテクノロジーのほか、大規模なシナリオを管理するには、複数の名前空間が必要になります。

New Relic.

別々のチームが同じ名前空間にプロジェクトをデプロイすると、お互いの作業に干渉し、影響が出る恐れがあります。別々の名前空間を設定することで、分離（アイソレーション）とチームベースのアクセスセキュリティにより、チームがお互いに干渉することなく、作業に集中できます。また、名前空間ごとにリソースの割り当てを設定できるので、リソースを大量に消費するアプリケーションがクラスタの容量を消費し、他のチームのリソースに影響を与えることはありません。

名前空間を使用する

クラスタを作成すると、Kubernetesはすぐに3つの名前空間を提供します。クラスタに付属する名前空間を一覧表示するには、次のコマンドを実行します。

```
$kubectl get namespaces
NAME            STATUS    AGE
default         ACTIVE    2d
kube-system     ACTIVE    2d
kube-public     ACTIVE    2d
```

`kube-system`名前空間は、Kubernetesエンジン用に確保されているもので、ユーザー用ではありません。`kube-public`名前空間は、クラスタ情報などのパブリックアクセスデータが格納される場所です。

デフォルトの名前空間は、アプリやサービスを作成する場所です。コンポーネントの作成で名前空間を指定しないと、Kubernetesはデフォルトの名前空間に作成します。しかし、デフォルトの名前空間が適しているのは、小規模なシステムの場合のみです。

Kubernetes名前空間は、以下の`kubectl`コマンドで作成します。

```
kubectl create namespace test
```

あるいは、YAML構成ファイルで名前空間を作成する方法もあります。クラスタで作成されたオブジェクトの履歴を構成ファイルのリポジトリに残しておきたい場合は、この方法が望ましいでしょう。以下の「`demo.yaml`」ファイルは、構成ファイルで名前空間を作成する方法を示します。

```
kind: Namespace
apiVersion: v1
metadata:
  name: demo
  labels:
    name: demo
kubectl apply -f demo.yaml
```

販売、請求、発送の3種類のプロジェクトのイメージを想像してください。先ほど述べた理由から、すべてをデフォルトの名前空間にデプロイしないよう、プロジェクトごとに名前空間を作成します。

重要なのは、プロジェクトにはそれぞれライフサイクルがあるため、開発リソースと生産リソースが混在しないようにすることです。つまり、プロジェクトが複雑になればなるほど、クラスタには高度な名前空間ソリューションが必要になります。さらに、クラスタを開発環境、ステージング環境、本番環境に分割することもできます。

```
kind: Namespace
apiVersion: v1
metadata:
  name: dev
  labels:
    name: dev
kubectl apply -f dev.yaml
```

```
kind: Namespace
apiVersion: v1
metadata:
  name: staging
  labels:
    name: staging
kubectl apply -f staging.yaml
```

```
kind: Namespace
apiVersion: v1
metadata:
  name: prod
  labels:
    name: prod
kubectl apply -f prod.yaml
```

プロジェクトの条件に応じて、採用可能な名前空間のリストは、以下のとおりです。

- sales-dev
- sales-staging
- sales-prod
- billing-dev
- billing-staging
- billing-prod
- shipping-dev
- shipping-staging
- shipping-prod

Kubernetesでリソースを作成する名前空間を指定する方法は2通りあります。

`kubectl` applyコマンドでリソースを作成する際、`namespace`フラグを指定します。

```
kubectl apply -f pod.yaml --namespace=demo
```

YAML構成ファイルのメタデータを変更し、宛先の名前空間属性を含める方法もあります。

```
apiVersion: v1
kind: Pod
metadata:
  name: testpod
  namespace: demo
```

```
  labels:
    name: testpod
spec:
  containers:
  - name: testpod
    image: nginx
```

YAML宣言で名前空間をあらかじめ定義しておくと、リソースは常にその名前空間で作成されるようになります。このリソースに別の名前空間を設定する`namespace`フラグを付けて`kubectl apply`コマンドを実行しようとすると、コマンドは失敗します。

ラベルを使用する

クラスタのオブジェクトの数が増加すると、検索や整理が難しくなります。プロジェクトが複雑になると、多次元的なコンポーネントが境界線を超え、固定されたクラスタ構造に問題が生じることになります。この場合は、ラベルを使用することで、クラスタのオブジェクトに内容を示すメタデータを添付できるので、オブジェクトの分類、検索、操作が一括で実行できるようになります。

ラベルは、オブジェクトに割り当てられたキー/値構造です。オブジェクトには、1つ以上のラベルペアを割り当てることができます。ラベルを添付しないという選択肢もあります。ラベルの用途：

- Podが本番環境とカナリア環境のどちらに含まれるかを判断する
- 安定版とアルファ版のリリースを区別する
- オブジェクトがどのレイヤー(UI、ビジネスロジック、データベースなど)に属するかを指定する
- Podがフロントエンドかバックエンドかを識別する
- オブジェクトのリリースバージョンを指定する(例：V1.0、V2.0、V2.1)

たとえば、以下の設定ファイルでは、layer: interfaceとversion: stableの2つのラベルを持つPodを定義しています。

```
apiVersion: v1
kind: Pod
metadata:
  name: app-gateway
  labels:
    layer: interface
    version: stable
```

ラベルを付加したら、定義した基準に従って、ラベルセレクタでKubernetesオブジェクトを選択することができます。

クラスタ内にいくつかのPodがあり、ラベルが割り当てられているとします。以下のコマンドで、すべてのPodとラベルが検索できます。

```
kubectl get pods --show-labels
```

```
NAME        READY STATUS   RESTARTS AGE LABELS
app-gateway 1/1   Running 0        1m
layer=interface,version=stable
```

```
micro-svc1   1/1    Running 0          1m
layer=business,version=stable
micro-svc2   1/1    Running 0          1m
layer=business,version=alpha
```

ラベルセレクタを使って、同じ`kubectl get pods`コマンドを実行すると、指定したラベルが添付されたPodのみが表示されます。以下のように`-L`セレクタを使用すれば、レイヤーのラベルのみを表示することができます。

```
kubectl get pods -L=layer
```

```
NAME         READY  STATUS  RESTARTS  AGE  LABELS
app-gateway  1/1    Running 0          1m   interface
micro-svc1   1/1    Running 0          1m   business
micro-svc2   1/1    Running 0          1m   business
```

結果を絞り込んで、interface podのみを検索したい場合は、-lを使用し、以下の条件を指定します。

```
kubectl get pods -l=layer=interface --show-labels
```

```
NAME         READY  STATUS   RESTARTS  AGE  LABELS
app-gateway  1/1    Running  0          1m
layer=interface,version=stable
```

ラベルとラベルセレクタについては、KubernetesLabels and Selectors（ラベルとセレクタ）ページをご覧ください。

アノテーション（注釈）を使用する

アノテーションは、ラベルに似ており、キーと値のペアで構成されていますが、検索に使用することを想定していない点だけがラベルとは異なります。

では、ラベルを使わず、アノテーションを使うのは何故でしょうか。

箱が積まれた倉庫を整理するところを想像してください。箱の外側には、小さなラベルが貼られており、識別、グループ化、配列に役立つ重要なデータが記載されています。次に、それぞれの箱に情報が入っていると考えてみてください。これらの内容物は、箱を開けると取り出せるアノテーションで、外からは見えないようになっています。

ラベルとは異なり、アノテーションはKubernetesオブジェクトの選択または識別に使用できないため、セレクタを使用してクエリを実行することはできません。このようなメタデータを外部のデータベースに格納することも可能ですが、これも複雑な作業です。その代わり、オブジェクト自体にアノテーションを付けた方が便利です。オブジェクトにアクセスすると、アノテーションを読むことができます。

アノテーションは、以下のようなさまざまなユースケースで役立ちます。

- Podのコンテナが格納されているDockerレジストリ
- コンテナのビルド元となるgitリポジトリ
- ロギング、監視、分析、または監査リポジトリの表示
- 名前、バージョン、ビルド情報などのデバッグ関連情報
- 他のエコシステムコンポーネントの関連オブジェクトのURLなどのシステム情報
- 設定やチェックポイントなど、ロールアウトのメタデータ

New Relic.

- コンポーネントのプロジェクト担当者の電話番号またはメールアドレス
- チームのウェブサイトのURL

以下の例では、Pod構成ファイルに、コンテナのビルド元となるgitリポジトリの情報と、チームマネージャーの電話番号が記載されています。

```
apiVersion: v1
kind: Pod
metadata:
  name: annotations-test
  annotations:
    repo: "https://git.your-big-company.com.br/lms/
new-proj"
    phone: 800-555-1212
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

`annotate`コマンドを使えば、既存のPodにアノテーションを追加することもできます。

```
kubectl annotate pod annotations-test
phone=800-555-1212
```

Podのアノテーションにアクセスする場合のコマンド：

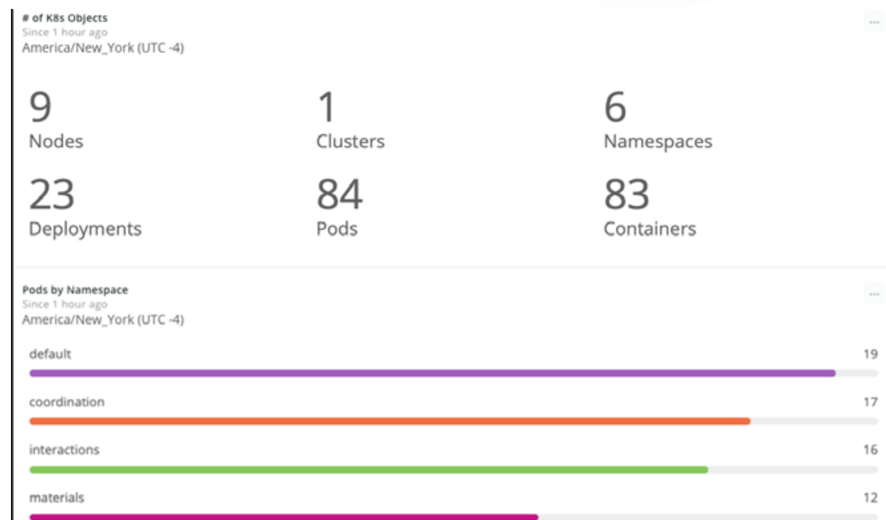
```
kubectl describe pod your-pod-name
```

これでPodに関する情報がすべてわかります。また、`JSONPath`テンプレートで`kubectl get pods`コマンドを使えば、Podからアノテーションデータのみを読み込むこともできます。

```
kubectl get pods -o=jsonpath="{.items[*]['metadata.
annotations']}"
```

クラスタを整理する、プロセスを整理する

名前空間、ラベル、アノテーションは、Kubernetesクラスタを整理して、管理するための便利なツールです。



Kubernetesの情報と名前空間別のPodの内訳を表示するNew Relic Oneのダッシュボード

いずれのツールも使い方は難しくありません。Kubernetesのコンセプトはスムーズに学べるものの、その対象は膨大です。

しかし、名前空間、ラベル、アノテーションを理解し、その使い方を理解できたので、Kubernetesへの道をさらに進めることができるはずです。

第5章：Kubernetesボリュームの操作

コンテナを使ったアプリケーションの実行には、さまざまなメリットがあります。しかし、格納のしやすさはそこに含まれていません。格納を実行するには、コンテナに一時的なファイルシステムが必要です。ただし、コンテナがシャットダウンすると、ファイルシステムへの変更がすべて失われます。簡単に交換できるコンテナには、持続性という概念が欠けるという副作用的な課題があります。

Dockerではこの問題をホストからのマウントポイントで解決しますが、Kubernetesの場合はさらに課題が発生する可能性があります。これまで学んできたように、Kubernetesにデプロイできる最小のコンピューティング単位は、Podです。状況によっては、Podの複数のインスタンスが、複数の物理的なマシンでホストされている場合があります。最悪な状況では、異なるコンテナが同じPodで実行されている場合でも、同じストレージにアクセスすることも考えられます。

本章では、ストレージの問題解決のため、Kubernetesが提供する2つのツール(ボリュームと永続ボリューム)について説明します。それぞれを使う理由と使い方を紹介します。

Kubernetesボリュームの概念

ボリュームとは、Podのすべてのコンテナ間で共有されるストレージです。ボリュームがあれば、同じPodで実行する複数のサービスで、同じマウント型ファイルシステムを使用することができます。ただし、この動作は自動ではありません。ボリュームが必要なコンテナは、どのボリュームを使用し、さらにコンテナのファイルシステムのどこにマウントするかを指定する必要があります。

さらに、ボリュームにはライフサイクルが明確に定義されており、所属するPodのライフサイクルに固定されています。Podがアクティブである限り、ボリュームも有効です。ただし、Podを再起動すると、ボリュームもリセットされます。この挙動を避けたい場合は、永続ボリューム（次のセクションで説明）を使用するか、希望に合わせてアプリケーションのロジックを変更する必要があります。

Kubernetesで注意するのはボリュームの正式な定義のみですが、実際は、現実（物理的）のファイルシステムを任意の場所に割り当てる必要があります。これが、DockerにはないKubernetesの利点です。Dockerは、ホストからコンテナへのパスをマッピングするだけですが、Kubernetesは、ストレージに適切なプロバイダーがあれば、基本的にあらゆるツールに対応し、

Amazon Elastic Block Store（EBS）やMicrosoft Azure Blob Storageなどのクラウドオプションや、Cephなどのオープンソースソリューションを使用できます。NFSのようなシンプルで汎用的なシステムを使用することもできます。Dockerのマウントパスに似たシステムを使いたい場合は、`hostPath`のボリュームタイプにフォールバックすることができます。

では、これらのボリュームはどのように作成するのでしょうか。Pod定義を使用します。

ボリュームの操作

たとえば、2つのコンテナ（いずれも、スリープ状態）を使って、sharedvolumeexampleという新しいPodを作成すると想定してみましょう。volumesキーを使用すると、コンテナ内で使用するボリュームを記述することができます。

```
kind: Pod
apiVersion: v1
metadata:
  name: sharedvolumeexample
spec:
  containers:
  - name: c1
    image: centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
    volumeMounts:
      - name: xchange
        mountPath: "/tmp/xchange"
  - name: c2
    image: centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
    volumeMounts:
      - name: xchange
        mountPath: "/tmp/data"
  volumes:
  - name: xchange
    emptyDir: {}
```

コンテナでボリュームを使用するには、上記のようにvolumeMountsを指定する必要があります。mountPathキーは、ボリュームアクセスパスを記述します。

2つのコンテナ間でボリュームを共有する様子を確認するため、テストを実行してみましょう。まず、仕様（sharedvolumeexample.ymlなど）からpodを作成します。

```
kubectl apply -f sharedvolumeexample.yml
```

次に、以下のコマンドで、最初のコンテナであるc1のターミナルにアクセスします。

```
kubectl exec -it sharedvolumeexample -c c1 -- bash
```

その後、/tmp/xchangeマウントポイントのファイルにデータを書き込みます。

```
echo 'some data' > /tmp/xchange/file.txt
```

別のターミナルを開き、c2というコンテナに接続します。

```
kubectl exec -it sharedvolumeexample -c c2 -- bash
```

ここでの違いは、/tmp/dataでマウントされたストレージからデータを読み込むということです。

```
cat /tmp/data/file.txt
```

この操作により、想定した通りの「一部のデータ」は生成されます。今度は、Podを削除します。

```
kubectl delete pod/sharedvolumeexample
```

永続ボリュームの操作

（通常の）ボリュームでは、ニーズを満たせない場合は、永続ボリュームに切り替えることができます。

永続ボリュームは、クラスタレベルに配置されるストレージオブジェクトです。つまり、永続ボリュームのライフサイクルは単体のPodではなく、クラスタそのものに拘束されます。永続ボリュームを使用すれば、Pod間でのデータ共有が可能になります。

永続ボリュームの利点は、単体のPodのコンテナだけでなく、複数のPod間で共有できることです。つまり、永続ボリュームはサイズを拡大すれば、規模を拡張できるのです。ただし、サイズの縮小はできません。

永続ボリュームには、通常のボリュームと同様、物理的なプロバイダーを選択するオプションがあります。ただし、プロビジョニングについては少し異なる点があります。

永続ボリュームをプロビジョニングする方法は2通りあります。

- **静的な方法：**すでにストレージ側ですべて割り当てているので、操作は必要ありません。バックグラウンドにある物理的なストレージは常に同じです。
- **動的な方法：**需要の増加に伴い、使用できるストレージの容量を増やしたいと思うかもしれません。需要は、ボリューム要求リソースを介して解決されます。これについては、後で説明します。動的なストレージのプロビジョニングを有効にするには、Kubernetes APIサーバーでDefaultStorageClass管理コントローラを有効にする必要があります。

拡張性の高いリソースにより、需要が拡大するシステムの場合は、動的なプロビジョニングの方が適しています。それ以外の場合は、シンプルな静的プロビジョニングの使用をお勧めします。

hostPathバックアップされたストレージに永続ボリュームを作成してみます。kindをPodとして構成するのではなく、PersistentVolumeとして構成することに注意してください。

New Relic.

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: persvolumeexample
  labels:
    type: local
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/data"
```

Podと同様に、これらのリソースは`kubectl`ツールを使って作成します。

kubectl apply -f persvolumeexample.yml

この例では、`persvolumeexample`という名前の新しい永続ボリュームを、最大ストレージ容量を10GBで作成しました。条件に応じてさまざまなアクセスモード（`Read-WriteOnce`、`ReadOnlyMany`、`ReadWriteMany`）が指定できますが、すべてのストレージプロバイダーで利用できるわけではありません。たとえば、AWS EBSは`ReadWriteOnce`のみをサポートします。

作成された永続ボリュームは、以下のリソースを介して利用することができます。`PersistentVolumeClaim`要求することで、十分なスペースが確保されます。これは、動的なプロビジョニング中に、Kubernetesが割り当てるスペースを広げようとすると、失敗する可能性があります。

3GBのプロビジョニングに対する要求を作成してみましょう。

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim-1
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

このプロビジョニングでは、`kubectl`を使用します。

```
kubectl apply -f myclaim-1.yml
```

このコマンドを実行すると、Kubernetesは要求に一致する永続ボリュームを検索します。この要求の使い方は簡単です。

```
kind: Pod
apiVersion: v1
metadata:
  name: volumeexample
```

```
spec:
  containers:
  - name: c1
    image: centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
    volumeMounts:
      - name: xchange
        mountPath: "/tmp/xchange"
  - name: c2
    image: centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
    volumeMounts:
      - name: xchange
        mountPath: "/tmp/data"
  volumes:
  - name: xchange
    persistentVolumeClaim:
      claimName: myclaim-1
```

今回の例と前の例を比較すると、volumesのセクションだけが変化し、他は何も変わっていません。

この要求では、ボリュームの一部のみを管理します。この部分を解除するには、要求を削除する必要があります。永続ボリュームの再要求ポリシーは、要求を解除した後のボリュームの処理をKubernetesに指定するものです。ここでの選択肢は、Retain、Recycle（動的プロビジョニングを優先するため、非推奨）、Deleteです。

再要求ポリシーを設定するには、Per-sistentVolume configのspecセクションで、persistent-VolumeReclaimPolicyオプションを定義する必要があります。たとえば、前の構成はこのようになります。

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: persvolumeexample
  labels:
    type: local
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: "/tmp/data"
```

ボリュームを賢く選択する

ボリュームと永続ボリュームの両方で、コンテナの再起動後も存続するデータストレージを追加することができます。ボリュームは Pod のライフサイクルに固定されますが、永続ボリュームは特定の Pod に依存せずに定義することができます。その後は、任意の Pod で使用することができます。

どれを選ぶかは、ニーズによって異なります。Pod がシャットダウンすると、ボリュームは削除されますが、Pod で実行するコンテナ間でデータを共有する必要がある場合に最適です。

永続ボリュームは各 Pod よりも寿命が長いため、Pod の再起動後も存続する必要がある、あるいは Pod 間でデータを共有する場合に最適です。

どちらのストレージも、クラスタで簡単にセットアップし、使用できます。

本書にまとめたアプローチを参考にしていただき、Kubernetes 環境をコントロールし、完璧なソフトウェアの出荷に近づくことを願っています。Kubernetes モニタリングについてさらなる詳細は、「A Complete Introduction to Monitoring Kubernetes with New Relic」をご覧ください。

Kubernetes Monitoring

クラスタ全体の可観測性を実現

Learn More