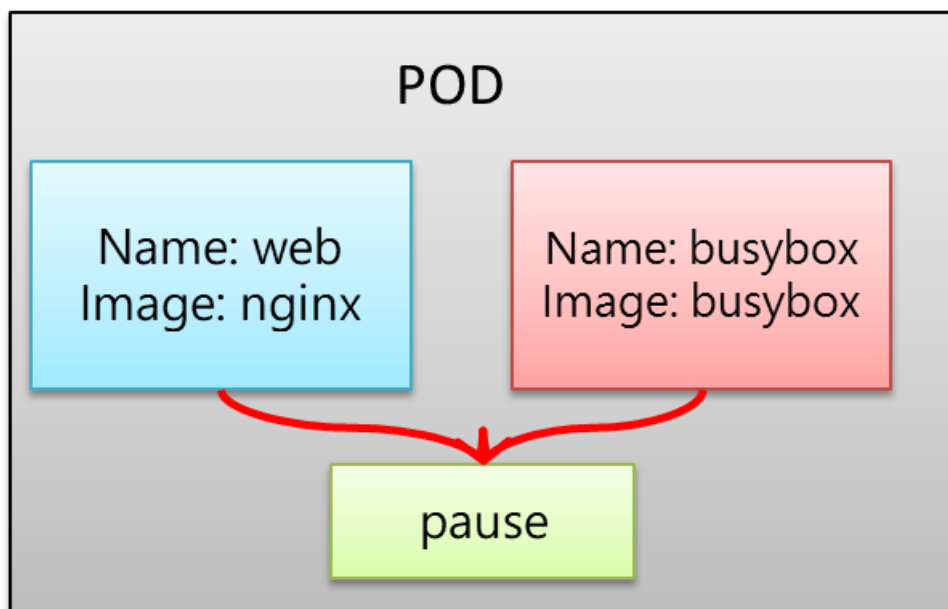
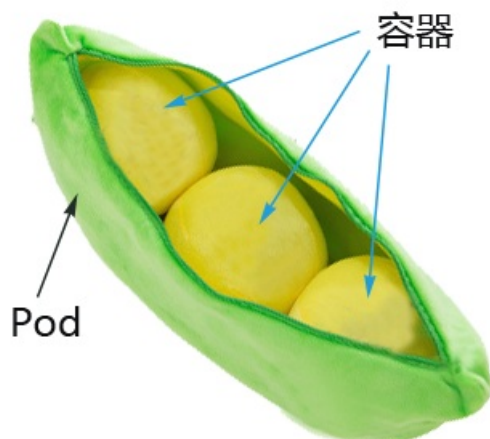


Pod 基本概念



Pod 概述

Pod 是 Kubernetes 系統中可以創建和管理的最小單位，是資源對象模型中由使用者創建或部署的最小資源對象模型，也是 Kubernetes 平台上運行容器化應用的資源對象。其他的資源對象都是用來支援或強化 Pod 的功能，例如：

- **控制器** 資源對象用來監控 Pod 的狀態
- **Service 或 Ingress** 資源對象用來暴露 Pod 引用外部訪問
- **PersistentVolume** 資源對象用來為 Pod 提供存儲等功能

Kubernetes 並不直接調度容器，而是調度 Pod。Pod 是由一個或多個容器組成的。

Pod 是 Kubernetes 的**最重要概念**。每一個 Pod 都有一個特殊的被稱為 **Pause 容器** 的容器。Pause 容器的鏡像對應於 Kubernetes 平台的一部分，負責維護 Pod 的網路命名空間及其他元數據。

除了 Pause 容器，每個 Pod 還包含一個或多個具備業務邏輯的使用者業務容器。

1. Pod 基本概念

1. 最小部署單位
2. 包含多個容器（容器集合）
3. 一個 Pod 中的容器共享網路命名空間
4. Pod 是短暫的（不永久存在）

2. Pod 存在意義

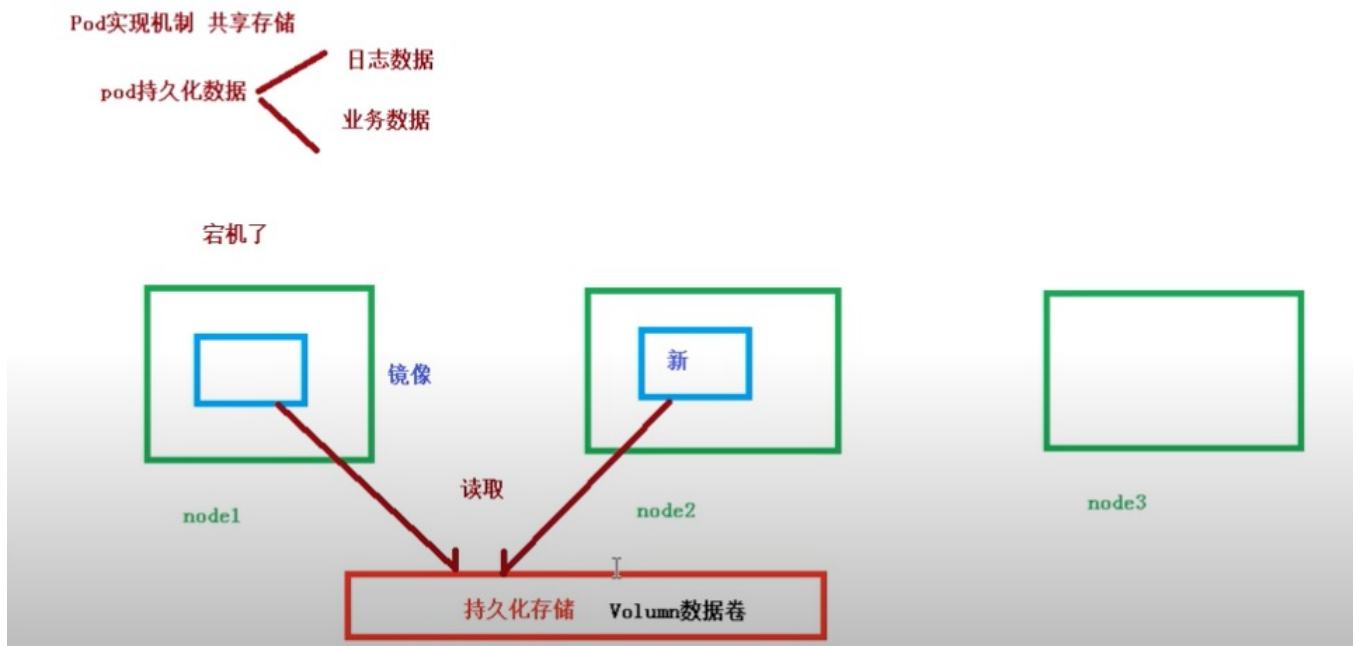
1. **創建容器使用 Docker**
 - 一個 Docker 對應一個容器
 - 一個容器有進程
 - 一個容器運行一個應用程序
2. **Pod 是多進程設計，運行多個應用程序**
 - 一個 Pod 有多個容器
 - 一個容器裡面運行一個應用程序
3. **Pod 存在為了親密性應用**
 - 兩個應用之間進行交互
 - 網路之間調用
 - 兩個應用需要頻繁調用

3. Pod 實現機制

1. 共享網路
2. 共享存儲

詳細說明：

- **共享網路：**
通過 Pause 容器，將其他業務容器加入到 Pause 容器裡面，讓所有業務容器在同一個命名空間中，可以實現網路共享。
- **共享存儲：**
引入數據卷的概念（Volume），使用數據卷進行持久化存儲。



以下是圖片中的文字辨識及繁體中文翻譯：

4. 鏡像拉取策略

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: nginx
      image: nginx:1.14
      imagePullPolicy: Always
```

鏡像拉取策略說明：

- **IfNotPresent**：預設值，當宿主機上沒有該鏡像時才會拉取。

- **Always**：每次創建 Pod 時都會重新拉取一次鏡像。
- **Never**：Pod 永遠不會主動拉取該鏡像。

以下是圖片中的文字辨識及翻譯成繁體中文：

5.Pod 資源限制示例

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
    - name: db
      image: mysql
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: "password"
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "500m"
```

Pod 和 Container 的資源請求與限制說明：

- `spec.containers[].resources.limits.cpu`：容器的 CPU 使用上限。
- `spec.containers[].resources.limits.memory`：容器的記憶體使用上限。
- `spec.containers[].resources.requests.cpu`：容器的 CPU 請求值。
- `spec.containers[].resources.requests.memory`：容器的記憶體請求值。

以下是圖片中的文字辨識及繁體中文翻譯：

6.Pod 重啟策略

```
apiVersion: v1
kind: Pod
metadata:
  name: dns-test
spec:
  containers:
    - name: busybox
      image: busybox:1.28.4
      args:
        - /bin/sh
        - -c
        - sleep 36000
      restartPolicy: Never
```

重啟策略說明：

- **Always**：當容器終止退出後，**總是重啟容器**。這是預設的策略。
- **OnFailure**：當容器異常退出（退出狀態碼非 0）時，**才重啟容器**。
- **Never**：當容器終止退出後，**不會重啟容器**。

以下是圖片中的文字辨識及繁體中文翻譯：

7.Pod 健康檢查

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
    name: liveness-exec
spec:
  containers:
    - name: liveness
      image: busybox
      args:
        - /bin/sh
        - -c
        - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy
      livenessProbe:
        exec:
          command:
            - cat
            - /tmp/healthy
          initialDelaySeconds: 5
          periodSeconds: 5
```

健康檢查說明：

- **livenessProbe（存活檢查）**
 - 如果檢查失敗，將**殺死容器**，根據 Pod 的 `restartPolicy` 來操作。
- **readinessProbe（就緒檢查）**
 - 如果檢查失敗，Kubernetes 會將 **Pod 從 Service 的 endpoints 中刪除**。
 - 以下是 `initialDelaySeconds` 和 `periodSeconds` 的詳細說明：

簡單範例說明：

```
initialDelaySeconds: 5
periodSeconds: 5
```

這表示：

1. Pod 啟動後等待 5 秒才開始執行健康檢查。
2. 之後每隔 5 秒執行一次健康檢查。

□ 其他常見健康檢查參數：

參數名稱	說明
<code>initialDelaySeconds</code>	首次檢查前的延遲時間
<code>periodSeconds</code>	每次檢查的間隔時間
<code>timeoutSeconds</code>	每次健康檢查的 超時時間
<code>successThreshold</code>	判定為 成功 所需的連續檢查次數
<code>failureThreshold</code>	判定為 失敗 所需的連續檢查次數

小結：

- `initialDelaySeconds`：控制首次健康檢查的延遲時間。
 - `periodSeconds`：控制每次健康檢查的間隔時間。
- 這些參數幫助 Kubernetes **有效管理容器的健康狀態**，從而避免過早或過頻的檢查導致誤判。
如果需要進一步解釋，請告訴我！☺

Probe 支援的三種檢查方法：

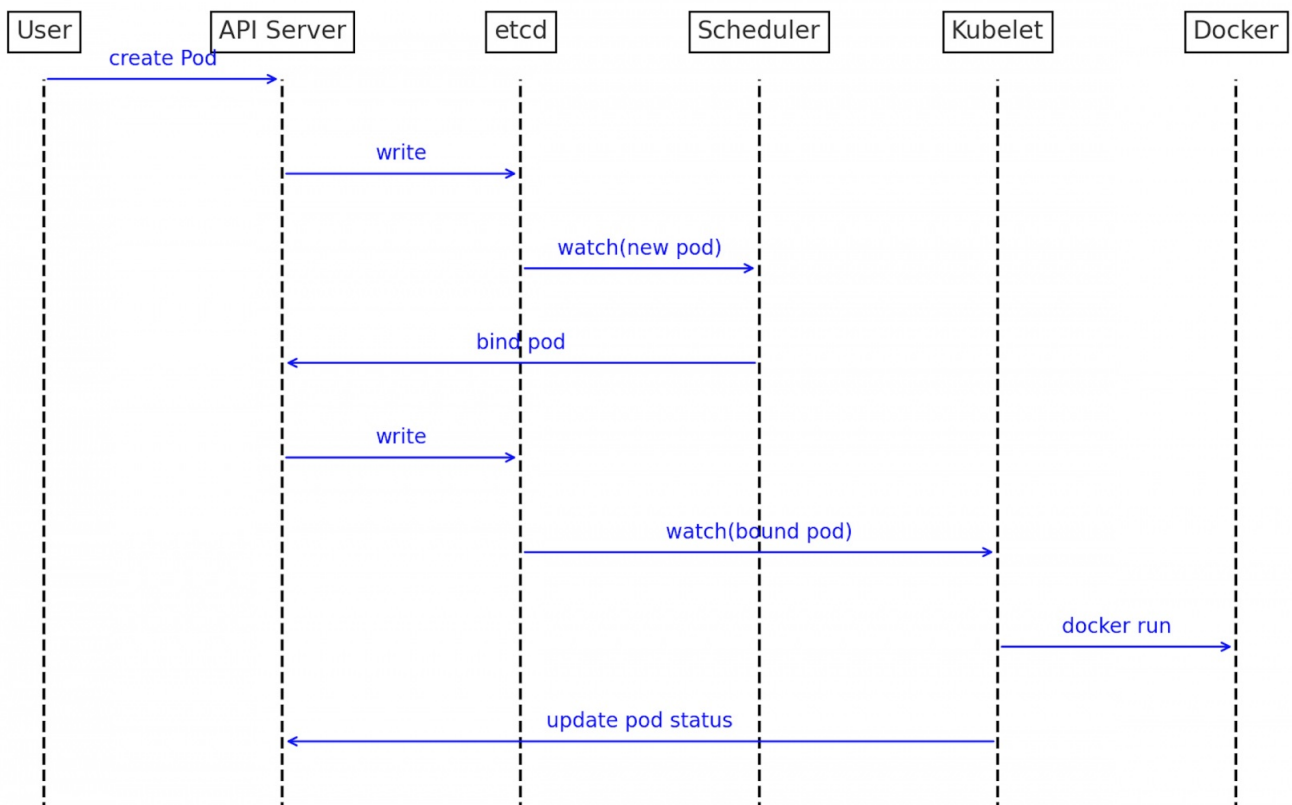
1. **httpGet**
 - 發送 HTTP 請求，返回 200-400 範圍的狀態碼為成功。
2. **exec**

- 執行 Shell 命令，返回狀態碼為 0 為成功。
3. **tcpSocket**
 - 發起 TCP Socket 建立成功。

Kubernetes Pod 創建流程圖

流程說明：

1. 使用者 向 **API Server** 發送 **create Pod** 的請求。
2. **API Server** 將 Pod 資訊 **寫入 (write)** 到 **etcd**。
3. **etcd** 監控到 **新的 Pod** 被創建 (**watch(new pod)**)，通知 **Scheduler**。
4. **Scheduler** 將 Pod **綁定 (bind pod)** 到節點。
5. **API Server** 再次將 **綁定資訊** 寫入 **etcd**。
6. **etcd** 監控到 **綁定的 Pod** (**watch(bound pod)**)，通知 **Kubelet**。
7. **Kubelet** 監控到 **綁定的 Pod**，執行 **docker run** 指令來啟動容器。
8. **API Server** 更新 **Pod 狀態**，並將 **狀態更新資訊** 寫入 **etcd**。



以下是圖片中的文字辨識及繁體中文翻譯：

8.Pod 調度

影響調用的屬性

1. Pod 資源限制對 Pod 調用產生影響

```
resources:
  requests:
    memory: "64Mi"
    cpu: "250m"
```

“ 根據 `requests` 找到足夠資源的節點來進行 Pod 調度。

2. 節點選擇器標籤對 Pod 調度的影響

```
spec:
  nodeSelector:
    env_role: dev
  containers:
  - name: nginx
    image: nginx:1.15
```

使用以下指令為節點打上標籤：

```
kubectl label node node1 env_role=prod
```

3. 節點親和性影響 Pod 調度

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: env_role
            operator: In
            values:
            - dev
            - test
        preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
            - key: group
              operator: In
              values:
              - otherprod
  containers:
  - name: webdemo
    image: nginx
```

節點親和性（`nodeAffinity`）說明

節點親和性和 `nodeSelector` 基本一樣，根據節點上的標籤條件來決定 Pod 調度到哪些節點上。

1. 硬親和性

- 使用 `requiredDuringSchedulingIgnoredDuringExecution`
- 條件必須滿足，否則無法調度到該節點。

2. 軟親和性

- 使用 `preferredDuringSchedulingIgnoredDuringExecution`
- **嘗試滿足條件，但不保證**。如果滿足條件的節點可用，則優先調度到這些節點。

重點區分

- **硬親和性**：必須滿足條件，否則調度失敗。
- **軟親和性**：優先考慮符合條件的節點，但不強制。