K8s

相關資源

Kubernetes in Action

https://fanatical-dentist-b1d.notion.site/Kubernetes-in-Action-8ac92f08a3fd41028ed1ae20cdc7a007

Kubernetes in Action2

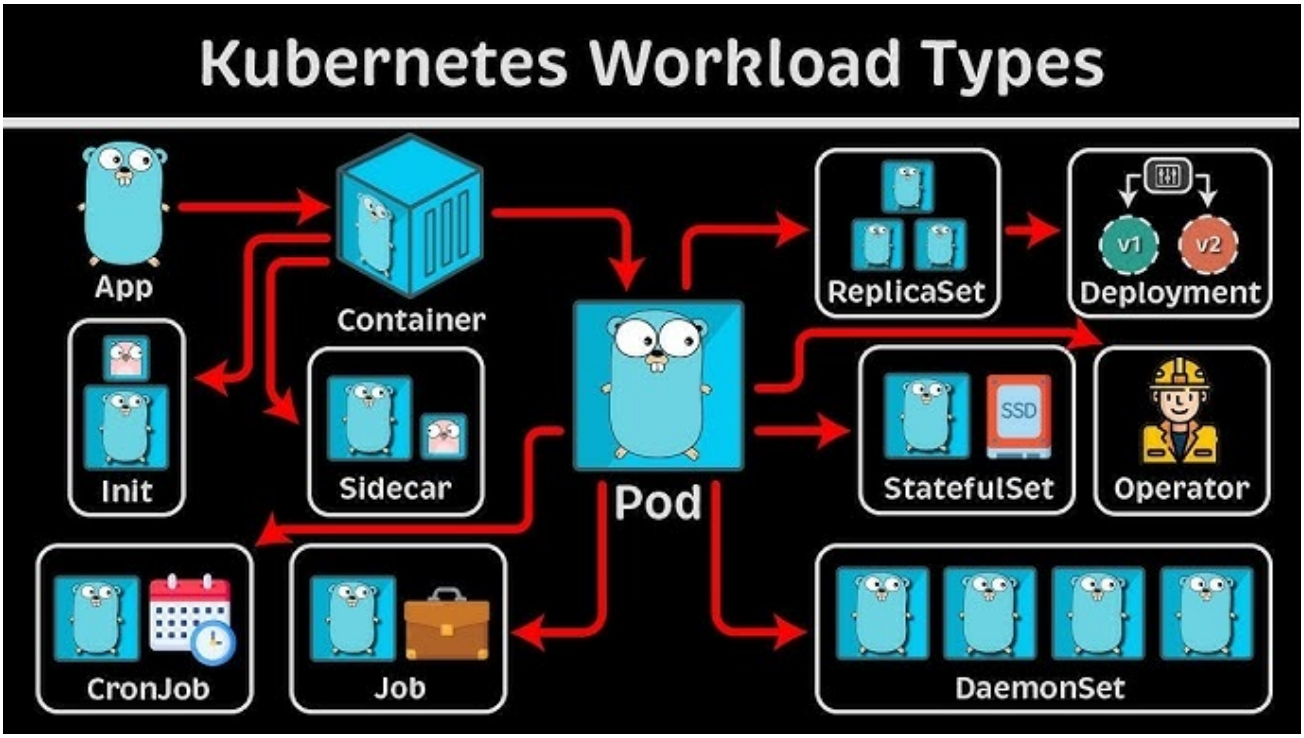
https://fanatical-dentist-b1d.notion.site/Kubernetes-in-Action2-4d565efc45124bc989484c8cdb7df181

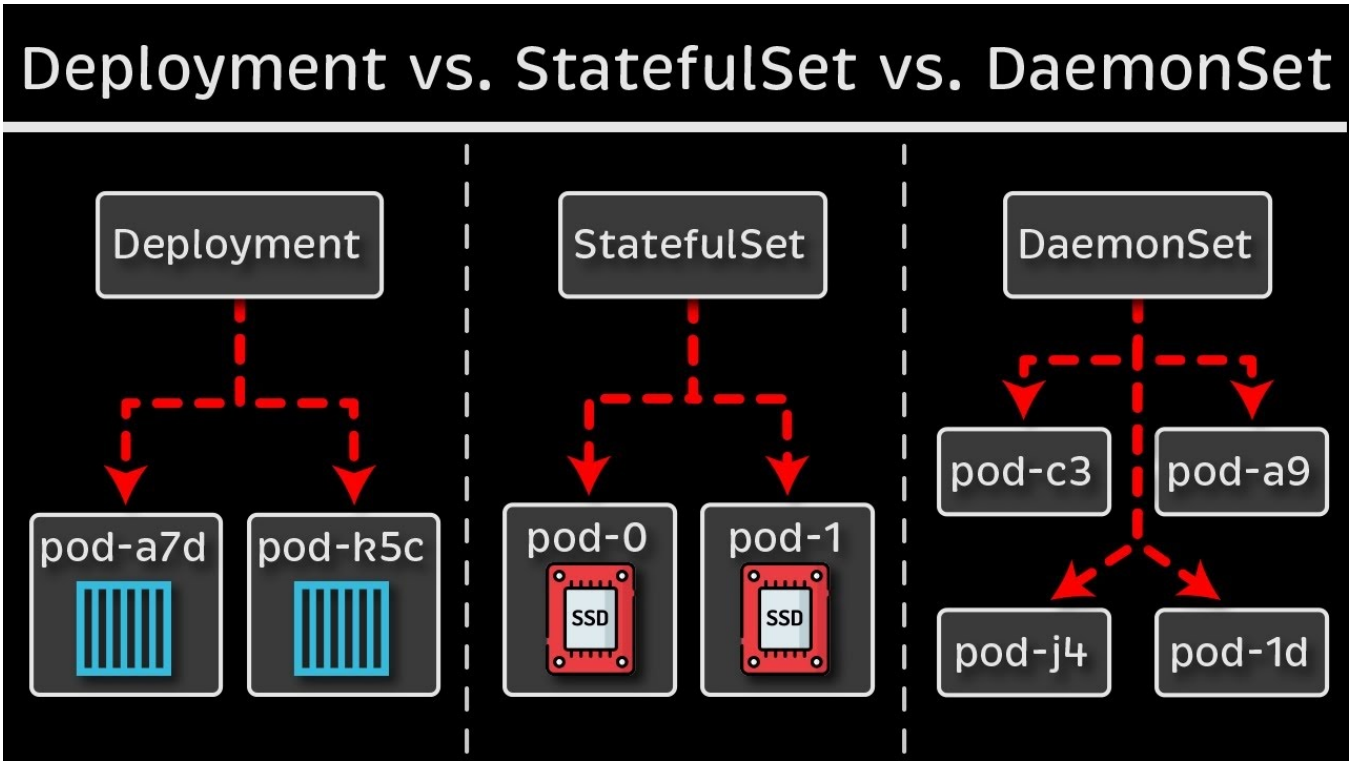
Kubernetes 官方文檔(簡中)

https://kubernetes.io/zh-cn/docs/home/

Kind 簡易說明

K8s Workload





以下是 **DaemonSet**、**StatefulSet**、**ReplicaSet** 和 **Deployment** 的詳細比較：

特性	DaemonSet	StatefulSet	ReplicaSet	Deployment
用途	確保每個 Node 上運行一個 Pod 副本。	部署有狀態的應用，每個 Pod 都有固定標識。	確保指定數量的 Pod 副本運行。	管理 Pod 和 ReplicaSet，提供滾動更新功能。
典型場景	日誌收集器、監控代理等需要全節點覆蓋。	資料庫、分佈式系統等需要穩定標識或存儲的應用。	確保無狀態應用的高可用性。	部署和管理無狀態應用，支持版本控制和更新。

特性	DaemonSet	StatefulSet	ReplicaSet	Deployment
Pod 名稱	名稱可變，與 Node 關聯，無固定格式。	有固定名稱（如 `db-0`, `db-1`）。	無固定格式，與 Deployment 類似。	通常由 Deployment 自動生成，不直接管理 Pod。
存儲	不涉及持久化存儲，通常為臨時存儲。	每個 Pod 可關聯專屬持久存儲。	通常無需存儲，支持臨時存儲。	通常無需存儲，支持無狀態應用。
縮放（副本數）	無法指定副本數，與 Node 數量相關。	支持指定副本數，且保證順序和標識穩定。	支持指定副本數，所有副本平等。	支持指定副本數，自動調整和滾動更新。
更新方式	滾動更新，逐一更新節點上的 Pod。	滾動更新，有序處理 Pod，確保依賴順序。	滾動更新，副本數量保持穩定。	滾動更新，支持回滾和藍綠部署策略。
依賴的資源	與 Node 關聯，不依賴其他資源。	通常依賴持久化存儲（如 PVC）。	通常不依賴其他資源。	自動創建和管理 ReplicaSet。
伸縮性	固定與 Node 數量匹配，無法手動調整。	可靈活擴展，並保證 Pod 的穩定標識。	手動指定副本數，無狀態，副本之間無區別。	自動管理副本數，支持水平和垂直擴展。
依賴順序	Pod 啟動順序無依賴。	Pod 啟動和刪除有順序，確保依賴關係。	無順序依賴，隨機啟動和刪除。	無順序依賴，通常關注應用級別的管理。
高可用性	和 Node 數量相關，通常與節點數一致。	通過固定標識和穩定的存儲保證高可用性。	通過多副本保證高可用性，但不管理應用版本。	提供高可用性並支持應用升級與版本回滾。

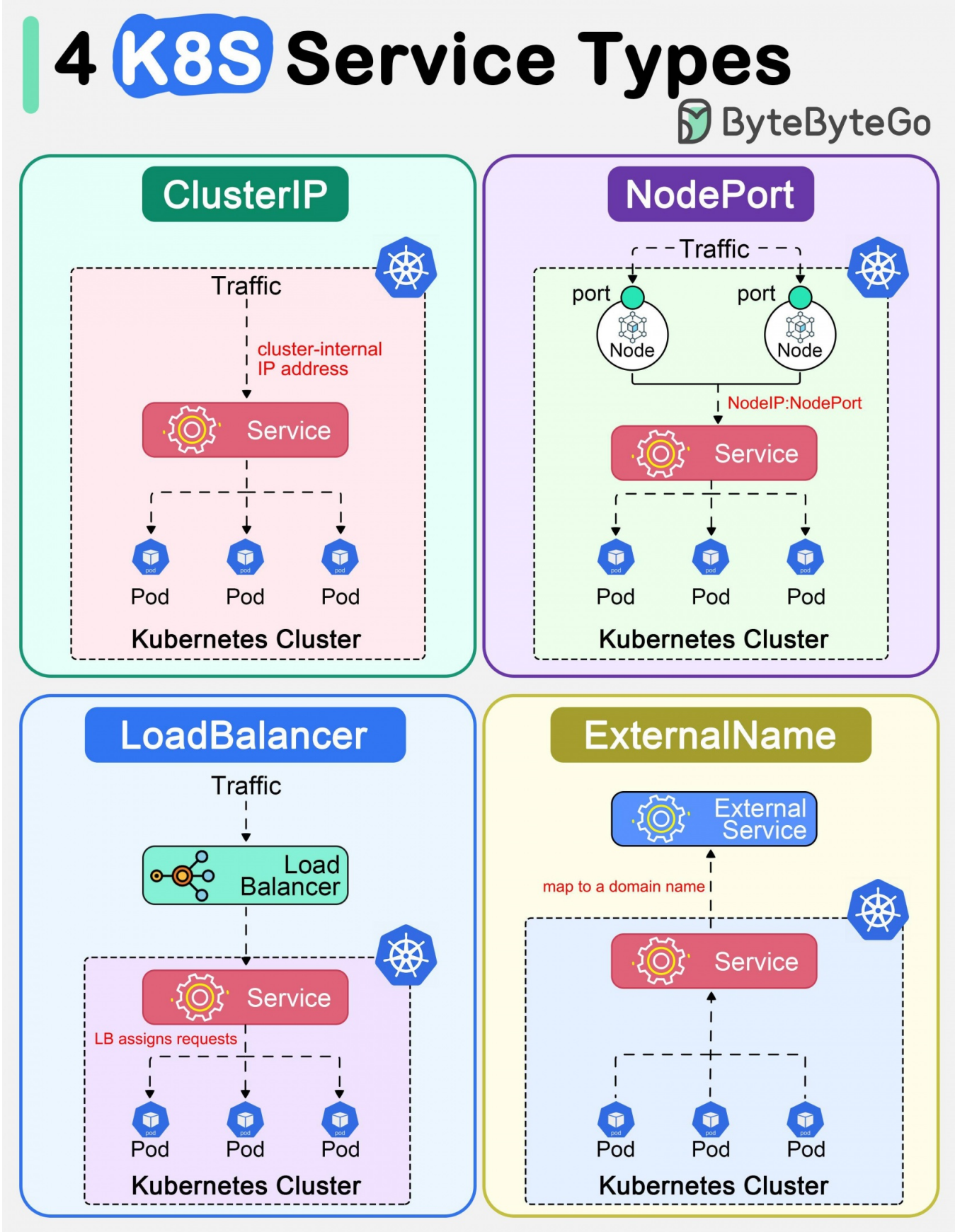
適用場景總結

1. **DaemonSet**：
   - 用於需要在所有 Node 上部署的服務，例如：
     - 日誌收集器（如 Fluentd）。
     - 監控代理（如 Prometheus Node Exporter）。
     - 網絡插件（如 Calico, Weave）。
2. **StatefulSet**：
   - 適用於需要穩定標識和存儲的有狀態應用，例如：
     - 資料庫（如 MySQL, PostgreSQL）。
     - 分布式系統（如 Kafka, ZooKeeper）。
     - 需要保持 Pod 順序的應用。
3. **ReplicaSet**：
   - 用於確保固定數量的無狀態 Pod 副本運行，例如：
     - 部署應用的核心部分，但一般由 Deployment 自動管理。
4. **Deployment**：
   - 最常用於管理無狀態應用，支持滾動更新、版本控制和自動擴展，例如：
     - Web 應用（如 Nginx, Node.js）。
     - 微服務應用的部署。

這樣的比較有助於釐清這些資源的功能和適用情境，幫助在實際項目中選擇合適的解決方案。

在 Kubernetes 中，`Service` 的 `type` 定義了服務的訪問方式和範圍。以下是 Kubernetes 支持的四種類型及其用途說明：

Service 的四種Type



1. ClusterIP (預設類型)

- **說明**：
  - 僅限於集群內部的虛擬 IP，用於服務發現和內部通信。
  - 外部無法直接訪問。
- **用途**：
  - 用於內部微服務之間的通信。
  - 通常搭配 DNS（如 `kube-dns`）進行名稱解析。
- **範例**：

```
apiVersion: v1
kind: Service
metadata:
  name: clusterip-service
spec:
  type: ClusterIP
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

---

2. **NodePort**

- **說明**：
  - 將服務暴露在每個節點的固定端口上（範圍：30000-32767）。
  - 用戶可以通過 `<NodeIP>:<NodePort>` 訪問服務。
- **用途**：
  - 測試環境中，用於將服務暴露給集群外部的用戶。
  - 生產環境中通常搭配 **Ingress** 或 **LoadBalancer**。
- **範例**：

```
apiVersion: v1
kind: Service
metadata:
  name: nodeport-service
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
      nodePort: 30007
```

---

3. **LoadBalancer**

- **說明**：
  - 在雲提供商（如 GCP、AWS 或 Azure）中創建外部負載均衡器，並將流量轉發到 Service。
  - 每個 LoadBalancer 類型的 Service 都會自動獲得一個外部 IP。
- **用途**：
  - 用於公開服務給互聯網，用於生產環境。
  - 支持流量負載均衡。
- **範例**：

```
apiVersion: v1
kind: Service
metadata:
  name: loadbalancer-service
spec:
  type: LoadBalancer
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

---

4. **ExternalName**

- **說明**：

    - 將 Service 名稱映射到外部 DNS 名稱。
    - 不會創建 ClusterIP，也無法進行內部流量轉發。
- **用途：**
    - 將 Kubernetes 內部服務請求代理到集群外部的服務。
- **範例：**

```
apiVersion: v1
kind: Service
metadata:
  name: externalname-service
spec:
  type: ExternalName
  externalName: external.example.com
```

服務類型對比

類型	是否對外暴露	訪問方式	典型使用場景
ClusterIP	否	集群內部通信，使用虛擬 IP	微服務之間的內部通信。
NodePort	是	`<NodeIP>:<NodePort>`	測試環境或簡單的外部訪問。
LoadBalancer	是	自動分配外部 IP	公開服務到互聯網，支持負載均衡。
ExternalName	是（間接）	使用外部 DNS 名稱解析	將內部服務請求代理到集群外部服務。

適用場景建議

1. **ClusterIP**：適合內部微服務的互相調用，默認類型。
2. **NodePort**：適合小型測試環境，簡單公開服務。
3. **LoadBalancer**：適合生產環境，需要高可用的負載均衡服務。
4. **ExternalName**：適合需要訪問外部資源但想使用 Kubernetes 內部 DNS 管理的情況。

這些 `type` 結合不同的網絡需求，靈活應對各種應用場景。

YML 的各種 Kind

Kind	說明	用途
Pod	Kubernetes 中最小的執行單位，包含容器。	部署單個應用程式容器。
Job	執行一次性任務，確保成功完成。	運行批處理任務。
CronJob	定期執行的任務。	排程任務（如每天備份數據）。
StatefulSet	管理有狀態的應用。	部署需固定標識或穩定存儲的 Pod。
ReplicaSet	確保指定數量的 Pod 始終運行。	通常由 Deployment 管理，用於維持 Pod 副本數。
DaemonSet	確保每個 Node 上運行一個 Pod 副本。	部署監控代理、日誌收集器等。
Deployment	管理 Pod 和 ReplicaSet，支持滾動更新。	部署無狀態應用，保持指定數量的 Pod 運行。
Service	定義 Pod 的網絡訪問規則。	將外部流量路由到 Pod，提供負載均衡。
Ingress	提供 HTTP/HTTPS 路由規則。	公開訪問服務，支持負載均衡和基於域名的路由。
NodePort	將外部流量映射到指定節點端口。	將集群外部訪問直接映射到內部 Service，通常為開發測試用途。

Kind	說明	用途
ConfigMap	存儲非機密配置數據。	將配置注入 Pod（環境變量或掛載文件）。
Secret	存儲機密數據（如密碼、API 密鑰）。	安全地將敏感信息注入 Pod。
PersistentVolume (PV)	定義持久化存儲。	提供存儲資源，獨立於 Pod 的生命週期。
PersistentVolumeClaim (PVC)	申請 PersistentVolume。	Pod 使用 PVC 來請求持久存儲資源。
Namespace	將資源分隔成不同的邏輯組。	支持多租戶環境。
Role 和 ClusterRole	定義訪問權限。	管控資源的讀寫權限，`Role` 限於 Namespace，`ClusterRole` 全集群。
RoleBinding 和 ClusterRoleBinding	綁定訪問權限到用戶。	授權用戶或服務帳號訪問權限。
HorizontalPodAutoscaler (HPA)	根據指標自動調整 Pod 副本數量。	提高應用的彈性擴展能力。

範例 YAML 說明

1. CronJob

執行每天凌晨 12 點的數據備份：

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: daily-backup
spec:
  schedule: "0 0 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: backup
            image: backup-tool:latest
            args:
            - /bin/sh
            - -c
            - "backup-script.sh"
          restartPolicy: OnFailure
```

2. DaemonSet

在每個 Node 上部署日誌收集器：

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: log-collector
spec:
  selector:
    matchLabels:
      app: log-collector
  template:
    metadata:
      labels:
```

```
      app: log-collector
  spec:
    containers:
    - name: log-collector
      image: log-collector:latest
```

---

3. Deployment

部署一個無狀態 Web 應用，副本數為 3：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web-app
  template:
    metadata:
      labels:
        app: web-app
    spec:
      containers:
      - name: web
        image: nginx:latest
        ports:
        - containerPort: 80
```

---

4. Job

執行一次性任務：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: one-time-task
spec:
  template:
    spec:
      containers:
      - name: task-runner
        image: busybox
        args:
        - /bin/sh
        - -c
        - "echo Hello, Kubernetes!"
      restartPolicy: OnFailure
```

---

5. StatefulSet

部署有狀態的數據庫應用：

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: database
spec:
  serviceName: "database-service"
  replicas: 3
  selector:
    matchLabels:
      app: database
  template:
```

```
    metadata:
      labels:
        app: database
    spec:
      containers:
      - name: db
        image: mysql:5.7
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: "password"
        ports:
        - containerPort: 3306
```

---

6. Pod

部署一個單一容器的應用：

```
apiVersion: v1
kind: Pod
metadata:
  name: single-app
spec:
  containers:
  - name: app
    image: busybox
    args:
    - /bin/sh
    - -c
    - "echo Running a single pod!"
```

1. CronJob

執行每天凌晨 12 點的數據備份：

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: daily-backup
spec:
  schedule: "0 0 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: backup
            image: backup-tool:latest
            args:
            - /bin/sh
            - -c
            - "backup-script.sh"
          restartPolicy: OnFailure
```

---

2. Service

定義一個 Service，將流量路由到 Pod：

```
apiVersion: v1
kind: Service
metadata:
  name: web-app-service
spec:
  selector:
    app: web-app
  ports:
    - protocol: TCP
```

```
    port: 80
    targetPort: 80
  type: ClusterIP
```

- **說明**: 這個 Service 將把進入集群內部的流量路由到標籤 `app: web-app` 的 Pod。

NodePort 範例

以下是一個簡單的 **NodePort** 配置範例，用於將集群外部的請求轉發到內部的 Pod：

```
apiVersion: v1
kind: Service
metadata:
  name: nodeport-service
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80        # ClusterIP 的服務端口
      targetPort: 8080 # Pod 內部容器的端口
      nodePort: 30007  # 節點上的固定外部端口（30000~32767 之間）
```

- **說明**：
  - `type: NodePort`：指定此服務為 NodePort 類型。
  - `nodePort`：設置固定的外部端口，供外部用戶通過 `<NodeIP>:30007` 訪問服務。
  - **適用場景**：開發測試環境或簡單的外部訪問，不建議用於生產環境，因為其功能有限且缺乏靈活性。

---

kubectl常用命令

`kubectl` 常用指令表格（繁體中文）

以下是 **`kubectl` 常用指令**的表格整理，包含了集群資訊查看、資源管理、排錯調試、日誌查看等各類常用操作，方便你快速查找和使用。

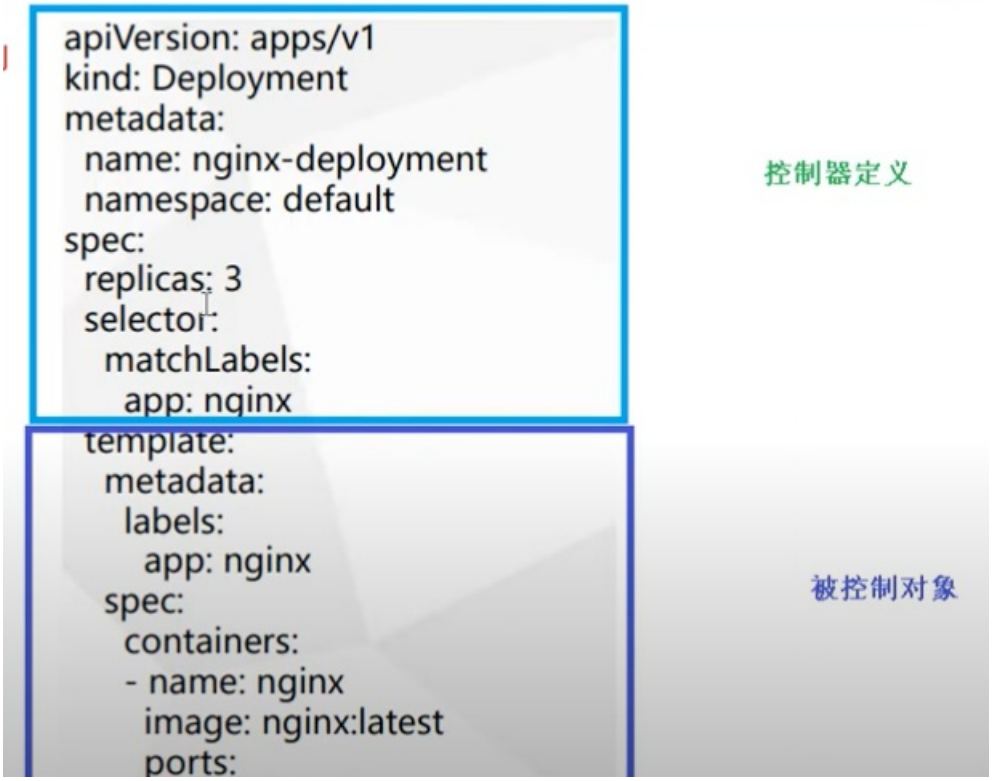
功能分類	指令	說明
集群資訊	`kubectl version`	查看客戶端和伺服器的版本資訊
	`kubectl cluster-info`	查看集群的基本資訊
	`kubectl get nodes`	列出所有節點（Nodes）
	`kubectl describe node <node>`	查看指定節點的詳細資訊
查看資源	`kubectl get pods`	查看目前命名空間的所有 Pods
	`kubectl get pods -n <namespace>`	查看指定命名空間的 Pods
	`kubectl get services`	查看目前命名空間的所有 Services
	`kubectl get deployments`	查看目前命名空間的所有 Deployments
	`kubectl get all`	查看目前命名空間的所有資源
查看詳細資訊	`kubectl describe pod <pod>`	查看 Pod 的詳細資訊
	`kubectl describe svc <service>`	查看 Service 的詳細資訊
創建資源	`kubectl apply -f <file.yaml>`	使用 YAML 文件創建或更新資源
	`kubectl create deployment`	創建一個新的 Deployment
更新資源	`kubectl edit <resource> <name>`	直接編輯現有資源
刪除資源	`kubectl delete pod <pod>`	刪除指定的 Pod
	`kubectl delete -f <file.yaml>`	根據 YAML 文件刪除資源
查看日誌	`kubectl logs <pod>`	查看指定 Pod 的日誌
	`kubectl logs <pod> -f`	即時追蹤 Pod 的日誌
	`kubectl logs <pod> --tail=100`	查看最近 100 行的 Pod 日誌
執行指令	`kubectl exec -it <pod> -- bash`	進入 Pod 的終端
	`kubectl exec <pod> -- <command>`	在 Pod 中執行指定的指令
排錯與調試	`kubectl describe pod <pod>`	查看 Pod 的詳細資訊，用於排查錯誤
	`kubectl get events`	查看集群的事件日誌
	`kubectl top node`	查看節點的資源使用狀況
	`kubectl top pod`	查看 Pod 的資源使用狀況
命名空間管理	`kubectl get ns`	查看所有命名空間
	`kubectl create namespace <ns>`	創建新的命名空間
	`kubectl delete namespace <ns>`	刪除指定的命名空間
配置管理	`kubectl config view`	查看當前的 kubeconfig 配置
	`kubectl config get-contexts`	查看可用的上下文（Contexts）
	`kubectl config use-context <ctx>`	切換至指定的上下文
YAML 輸出	`kubectl get <resource> -o yaml`	將資源輸出為 YAML 格式
監控資源	`kubectl get pods --watch`	即時監控 Pod 的狀態變化
滾動更新	`kubectl rollout status <deploy>`	查看 Deployment 的滾動更新狀態
	`kubectl rollout undo <deploy>`	回滾 Deployment 至上一版本

```
kubectl create deployment nginx --image=nginx
kubectl expose deployment --port=80 --type=NodePort
kubectl get pod,svc
```

```
kubectl get cs
kubectl get pods
kubectl apply -f
```

yaml 說明

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15
        ports:
        - containerPort: 80
```



欄位名稱	說明
apiVersion	API 版本
kind	資源類型
metadata	資源元數據
spec	資源規格
replicas	副本數量
selector	標籤選擇器
template	Pod 模板
metadata	Pod 元數據
spec	Pod 規格
containers	容器配置

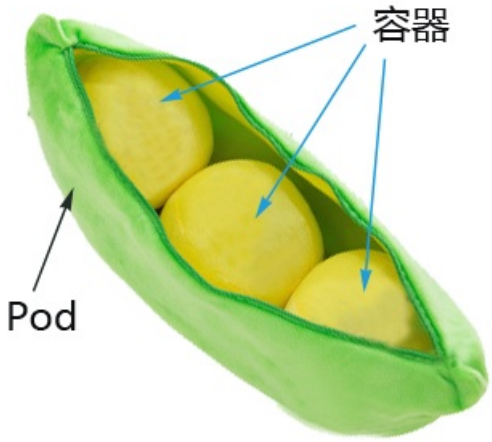
使用kubectl create 生成 yaml 文件

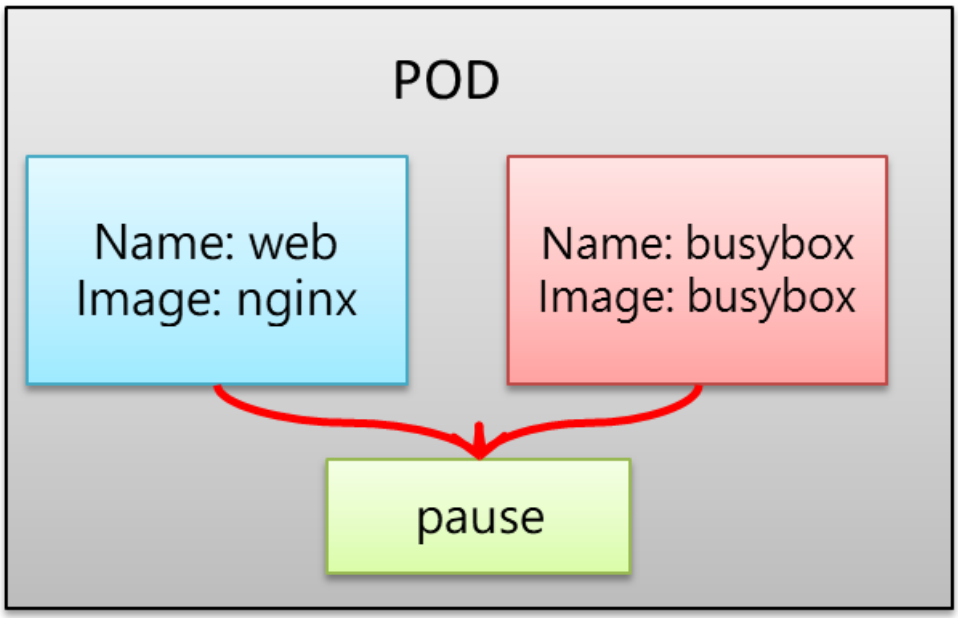
```
kubectl create deployment web --image=nginx -o yaml --dry-run > my1.yaml
```

使用kubectl get 導出 yaml 文件

```
kubectl get deploy niginx -o=yaml --export > my2.yaml
```

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

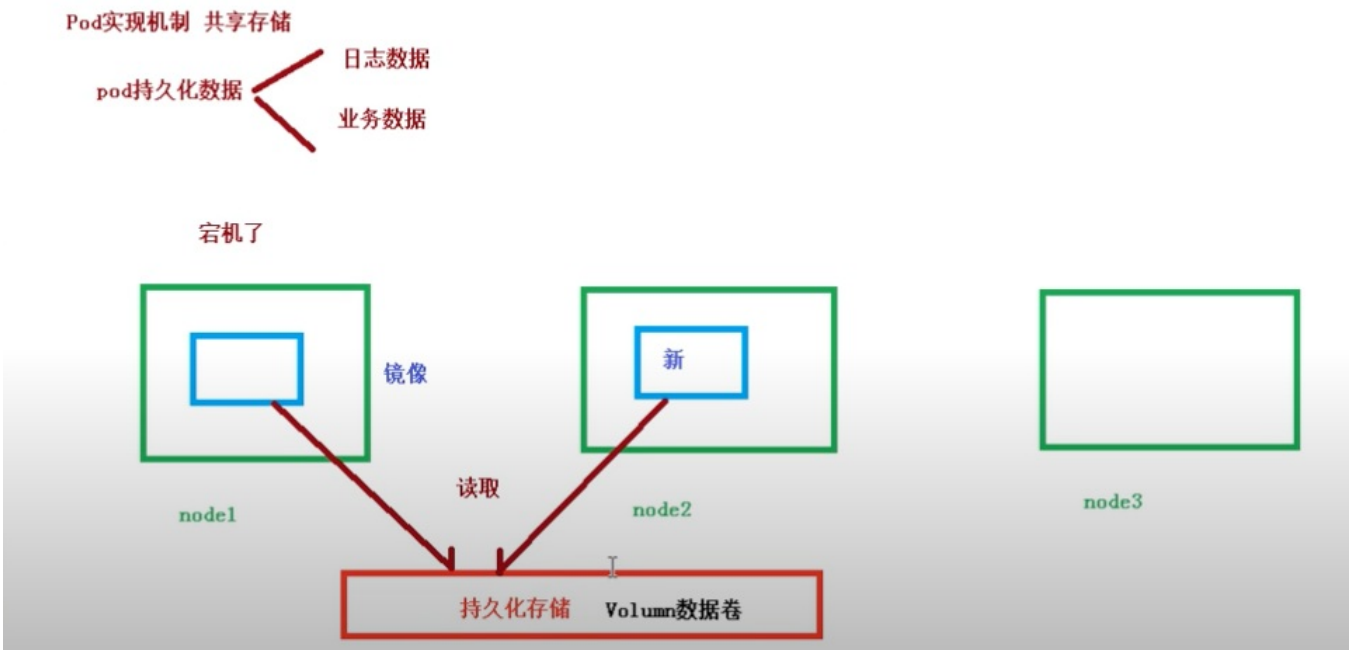
1. **創建容器使用 Docker**
   - 一個 Docker 對應一個容器
   - 一個容器有進程
   - 一個容器運行一個應用程序
2. **Pod 是多進程設計，運行多個應用程序**
   - 一個 Pod 有多個容器
   - 一個容器裡面運行一個應用程序
3. **Pod 存在為了親密性應用**
   - 兩個應用之間進行交互
   - 網路之間調用
   - 兩個應用需要頻繁調用

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

```yaml
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：

```yaml
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
`initialDelaySeconds`	首次檢查前的延遲時間
`periodSeconds`	每次檢查的間隔時間
`timeoutSeconds`	每次健康檢查的**超時時間**
`successThreshold`	判定為**成功**所需的連續檢查次數
`failureThreshold`	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**

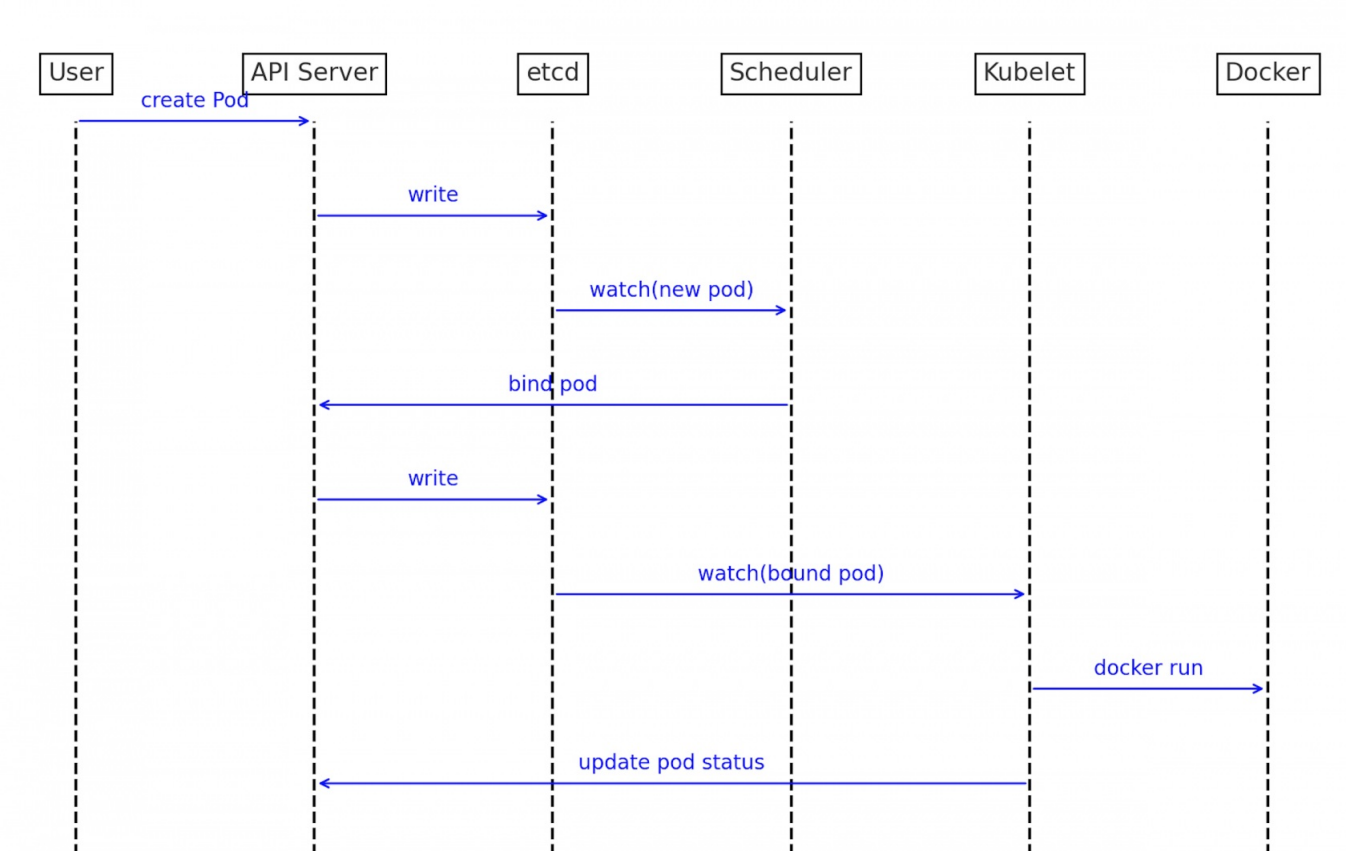
- 執行 Shell 命令，返回狀態碼為 0 為成功。

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。

Controller 基本概念

1. 什麼是 Controller

- 在集群上管理和運行容器的對象

---

2. Pod 和 Controller 的關係

- **Pod** 是通過 **Controller** 實現應用的運維，例如 **伸縮**、**滾動升級** 等
- **Pod** 和 **Controller** 之間通過 **label** **標籤** 建立關係 (**selector**)

---

3. Deployment 的應用場景

- 部署 **無狀態應用**
- 管理 **Pod** 和 **ReplicaSet**
- 支援 **滾動升級**、**回滾**、**彈性伸縮** 等功能

應用場景：

- **Web** **服務**
- **微服務**

以下是圖片中的文字辨識及繁體中文翻譯：

以下是圖片中的文字辨識及繁體中文翻譯：

---

4. 使用 Deployment 部署應用（YAML）

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: web
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: web
```

說明：

- **selector**：用於選擇符合特定標籤（matchLabels）的 Pod。
- **matchLabels**：指定要匹配的標籤條件，此處為 app: web。
- **labels**：在模板中定義 Pod 的標籤，確保與 selector 中的標籤條件一致。

此 YAML 文件表示使用 Deployment 部署一個副本的應用，並將其標籤設置為 app: web，以便進行管理和選擇。

---

5. 應用升級回滾和彈性伸縮

應用升級

```
[root@k8smaster ~]# kubectl set image deployment web nginx=nginx:1.15
deployment.apps/web image updated
```

查看升級狀態

```
[root@k8smaster ~]# kubectl rollout status deployment web
deployment "web" successfully rolled out
```

查看升級版本

```
[root@k8smaster ~]# kubectl rollout history deployment web
deployment.apps/web
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

回滾到上一個版本

```
[root@k8smaster ~]# kubectl rollout undo deployment web
deployment.apps/web rolled back
[root@k8smaster ~]# kubectl rollout status deployment web
Waiting for deployment "web" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "web" rollout to finish: 1 old replicas are pending termination...
deployment "web" successfully rolled out
```

回滾到指定的版本

```
[root@k8smaster ~]# kubectl rollout undo deployment web --to-revision=2
deployment.apps/web rolled back
```

```
[root@k8smaster ~]# kubectl rollout status deployment web
Waiting for deployment "web" rollout to finish: 1 out of 2 new replicas have been updated...
Waiting for deployment "web" rollout to finish: 1 old replicas are pending termination...
deployment "web" successfully rolled out
```

說明：

- 使用 `--to-revision=2` 指定要回滾到的版本號。
- **`kubectl rollout status`** 用於查看回滾的進度，確認部署是否成功完成。

以下是圖片中的文字辨識及繁體中文翻譯：

彈性伸縮

```
[root@k8smaster ~]# kubectl scale deployment web --replicas=10
deployment.apps/web scaled
```

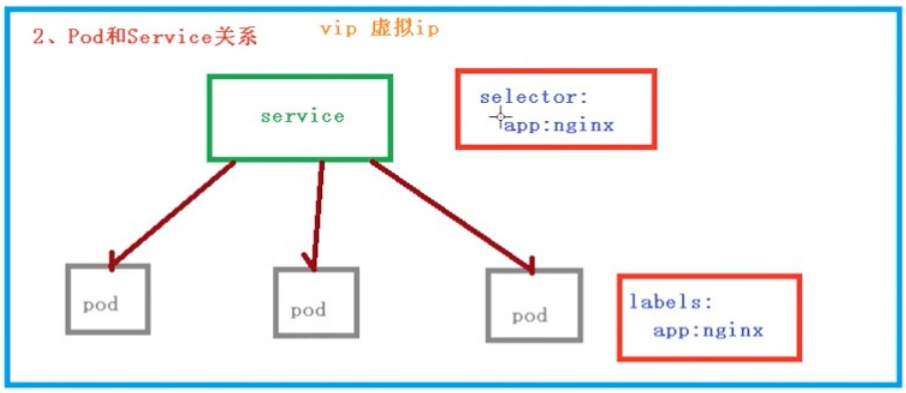
說明：

- 使用 `kubectl scale` 指令對 Deployment 進行彈性伸縮。
- **`--replicas=10`** 表示將 Pod 副本數量調整為 10。

此操作用於根據業務需求動態調整應用的容器數量，實現資源的彈性管理。

Service 基本概念

服務發現



根据label和selector
标签建立关联的

通过service实现Pod的负载均衡

以下是圖片內容的文字辨識及繁體中文翻譯：

---

1. 無狀態和有狀態

(1) 無狀態：

- 認為 **Pod** 都是一樣的
- 沒有順序要求
- 不用考慮在哪個 **Node** 運行
- 可以隨意進行伸縮和擴展

(2) 有狀態：

- 上述因素都需要考慮到
- 讓每個 **Pod** 獨立，保持 **Pod** 啟動順序和唯一性
- 唯一的網路標識符，持久存儲
- 有順序，例如 **MySQL 主從**

---

2. 部署有狀態應用

(1) 無頭 Service

- **ClusterIP: None**

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
    - port: 80
      name: web
  clusterIP: None
  selector:
    app: nginx
```

---

(2) StatefulSet 部署有狀態應用

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx-statefulset
  namespace: default
```

指令：

```
[root@k8smaster ~]# vi sts.yaml
[root@k8smaster ~]# kubectl apply -f sts.yaml
service/nginx created
statefulset.apps/nginx-statefulset created
```

說明：

- **無頭 Service**：`ClusterIP: None` 用於有狀態應用，允許 Pod 之間直接通訊。
- **StatefulSet**：適用於有狀態應用，確保每個 Pod 都有唯一標識，並保證啟動順序。

以下是圖片內容的文字辨識及繁體中文翻譯：

查看 Pod

- 有三個 Pod，每個都有唯一名稱：

```
[root@k8smaster ~]# kubectl get pods
NAME              READY   STATUS    RESTARTS   AGE
nginx-statefulset-0    1/1    Running   0        74s
nginx-statefulset-1    1/1    Running   0        40s
nginx-statefulset-2    1/1    Running   0        21s
```

Deployment 和 StatefulSet 的區別

- StatefulSet 有身份的（唯一標識的）。
- 根據主機名 + 按照一定規則生成域名。

查看創建的無頭 Service

```
NAME     TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)      AGE
kubernetes   ClusterIP   10.96.0.1    <none>       443/TCP      6d17h
nginx        ClusterIP   None         <none>       80/TCP       2m12s
```

```
web         NodePort    10.96.18.7    <none>       80:31819/TCP   50m
```

- **每個 Pod 有唯一主機名**
- **唯一域名**

格式：

```
主機名稱.service名稱.命名空間.svc.cluster.local
```

示例：

```
nginx-statefulset-0.nginx.default.svc.cluster.local
```

說明：

- StatefulSet 通過主機名和域名，確保每個 Pod 都有唯一標識並支持有狀態應用。
- 無頭 Service 的 ClusterIP 為 `None`，允許 Pod 之間直接通信。

以下是圖片內容的文字辨識及繁體中文翻譯：

---

3. 部署守護進程 DaemonSet

- **在每個 Node 上運行一個 Pod，新加入的 Node 也同樣運行在一個 Pod 裡面。**

示例：

在每個 Node 節點安裝數據採集工具。

指令操作：

1. **部署 DaemonSet：**

```
[root@k8smaster ~]# kubectl apply -f ds.yaml
daemonset.apps/ds-test created
```

2. **查看 Pod：**

```
[root@k8smaster ~]# kubectl get pods
NAME              READY   STATUS              RESTARTS   AGE
ds-test-cbk6v     0/1     ContainerCreating   0          0s
ds-test-cx6fk     1/1     Running             0          30s
```

3. **進入 Pod：**

```
[root@k8smaster ~]# kubectl exec -it ds-test-cbk6v bash
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in future versions.
```

4. **查看文件：**

```
root@ds-test-cx6fk:/# ls /tmp/log
anaconda   boot.log-20200902   containers   firewalld
```

說明：

- **DaemonSet** 用於確保每個 Node 上都執行一個 Pod，例如安裝系統監控、日誌收集、數據採集等工具。
- **特點：** 新加入的 Node 會自動部署相應的 Pod，適合全局性的守護進程需求。

ConfigMap

以下是圖片內容的辨識及繁體中文翻譯，並補充完整 **1-4 步驟** 的詳細說明：

ConfigMap 操作步驟

1. 創建配置文件

創建一個配置文件，例如 `redis.properties`：

```
[root@k8smaster ~]# vi redis.properties
# 文件內容
redis.host=127.0.0.1
redis.port=6379
redis.password=123456
```

---

2. 創建 ConfigMap

將配置文件創建為 ConfigMap：

```
[root@k8smaster ~]# kubectl create configmap redis-config --from-file=redis.properties
configmap/redis-config created

[root@k8smaster ~]# kubectl get cm
NAME           DATA   AGE
redis-config   1      10s
```

檢查 ConfigMap 詳情：

```
[root@k8smaster ~]# kubectl describe cm redis-config
Name: redis-config
Data:
redis.properties:
  redis.host=127.0.0.1
  redis.port=6379
  redis.password=123456
```

---

3. 以 Volume 形式挂載到 Pod 容器中

步驟：

1. 編寫 YAML 文件，將 ConfigMap 作為 Volume 注入到 Pod 中：

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mycontainer
    image: busybox
    command: ["sh", "-c", "cat /config/redis.properties; sleep 3600"]
    volumeMounts:
    - name: config-volume
      mountPath: /config
  volumes:
  - name: config-volume
    configMap:
      name: redis-config
```

2. 應用配置：

```
[root@k8smaster ~]# kubectl apply -f cm.yaml
pod/mypod created
```

3. 查看 Pod 狀態並檢查配置：

```
[root@k8smaster ~]# kubectl get pods
NAME       READY   STATUS      RESTARTS   AGE
mypod      1/1     Completed   0          41s

[root@k8smaster ~]# kubectl logs mypod
redis.host=127.0.0.1
redis.port=6379
redis.password=123456
```

---

4. 以變量形式挂載到 Pod 容器中

步驟：

1. 編寫 YAML 文件，將 ConfigMap 值作為環境變量注入：

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mycontainer
    image: busybox
    command: ["sh", "-c", "echo $REDIS_HOST; echo $REDIS_PORT; echo $REDIS_PASSWORD; sleep 3600"]
    env:
    - name: REDIS_HOST
      valueFrom:
        configMapKeyRef:
          name: redis-config
          key: redis.host
    - name: REDIS_PORT
      valueFrom:
        configMapKeyRef:
          name: redis-config
          key: redis.port
    - name: REDIS_PASSWORD
      valueFrom:
        configMapKeyRef:
          name: redis-config
          key: redis.password
```

2. 應用配置：

```
[root@k8smaster ~]# kubectl apply -f myconfig.yaml
pod/mypod created
```

3. 查看 Pod 狀態並檢查日誌：

```
[root@k8smaster ~]# kubectl logs mypod
127.0.0.1
6379
123456
```

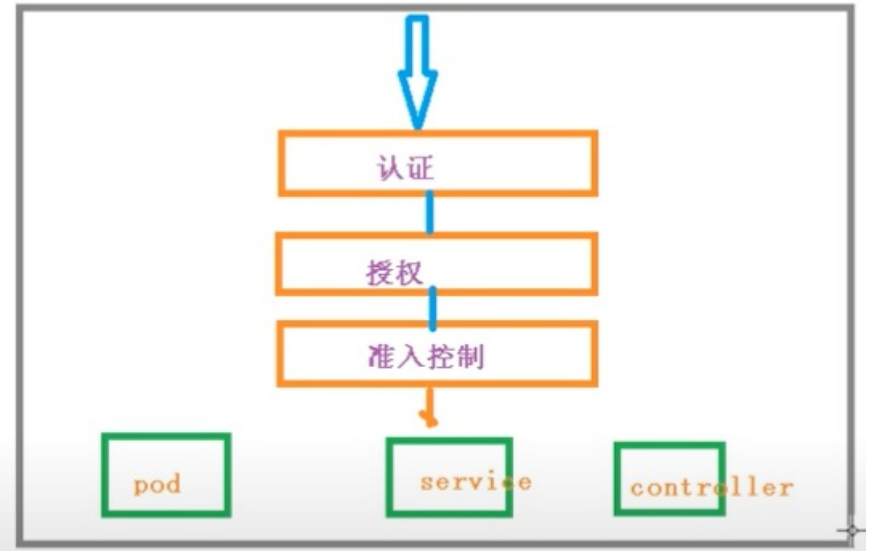
小結

- **Volume 挂載**：將 ConfigMap 文件直接映射到容器的文件系統中。
- **環境變量注入**：將 ConfigMap 的鍵值對作為環境變量傳遞給容器。

K8s 安全機制

以下是圖片的文字辨識及繁體中文翻譯：

Kubernetes 集群安全機制



1. 概述

1. **訪問 k8s 集群時**，需要經過三個步驟完成具體操作：
   - 第一步：認證
   - 第二步：授權
   - 第三步：準入控制
2. **訪問過程：**
   - 訪問時需要經過 `apiserver`，`apiserver` 做統一認證和驗證，例如門衛。
   - 訪問過程中需要 **證書、token**，或者用 **用戶名+密碼**。
   - 如果訪問 Pod，還需要 **serviceAccount**。

第一步 認證：傳輸安全

- **傳輸安全：**
  - 對外不暴露 8080 端口，只能內部訪問，對外使用端口 6443。
- **認證方式：**
  - `https` 證書認證：基於 CA 證書。
  - `http` token 認證：通過 token 識別用戶。
  - `http` 基本認證：使用用戶名+密碼認證。

第二步 授權：

- 基於 RBAC 進行授權操作。
- 基於角色訪問控制。

第三步 準入控制：

- 準入控制器的列表，根據列表決定是否允許執行操作。

---

2. RBAC

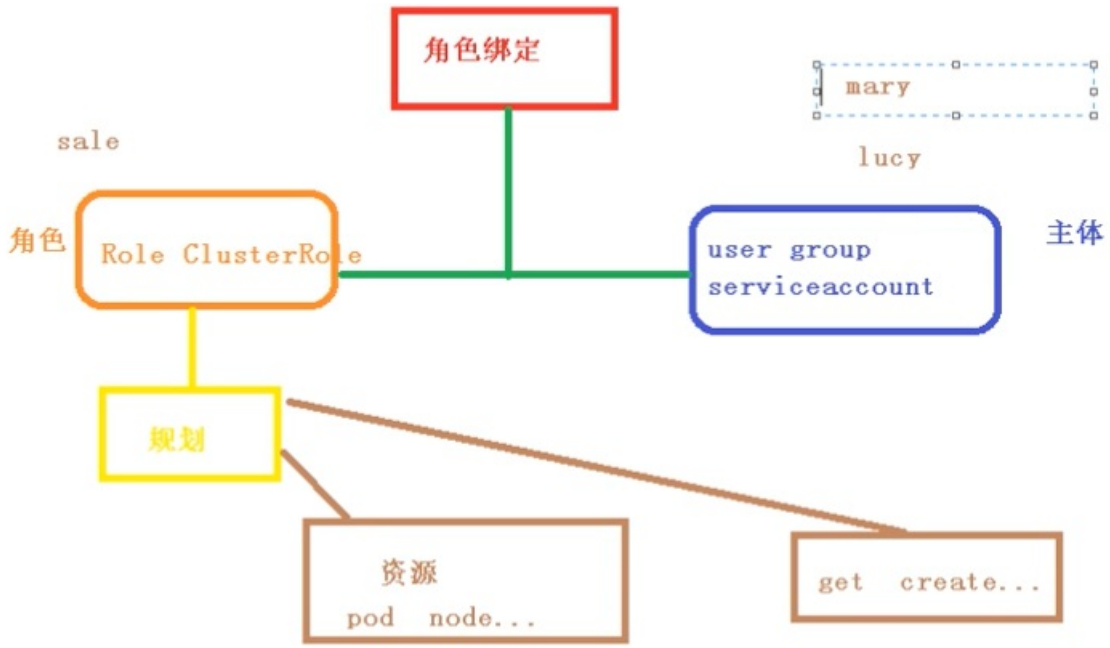
基於角色的訪問控制

- **角色：**
  - **role**：特定命名空間訪問權限
  - **ClusterRole**：所有命名空間訪問權限
- **角色綁定：**
  - **roleBinding**：角色綁定到主體
  - **ClusterRoleBinding**：集群角色綁定到主體

主體：

- **user**：用戶
- **group**：用戶組
- **serviceAccount**：服務帳號

這是基於 RBAC 的角色和權限管理，適用於 Kubernetes 的訪問控制。如需進一步說明或具體範例，請隨時告訴我！😊



以下是圖片中的文字辨識及翻譯成繁體中文：

---

1. 創建命名空間

```
[root@m1 lucy]# kubectl create ns roledemo
namespace/roledemo created
```

---

2. 在新創建的命名空間中創建 Pod

```
[root@m1]# kubectl run nginx --image=nginx -n roledemo
```

---

3. 創建角色

```
[root@m1]# vi rbac-role.yaml
[root@m1]# kubectl apply -f rbac-role.yaml
role.rbac.authorization.k8s.io/pod-reader created

[root@m1]# kubectl get role -n roledemo
NAME         AGE
pod-reader   19s
```

---

4. 創建角色綁定

```
[root@m1]# vi rbac-rolebinding.yaml
[root@m1]# kubectl apply -f rbac-rolebinding.yaml
rolebinding.rbac.authorization.k8s.io/read-pods created

[root@m1]# kubectl get rolebinding -n roledemo
NAME          AGE
read-pods     15s
```

5. 使用證書識別身份

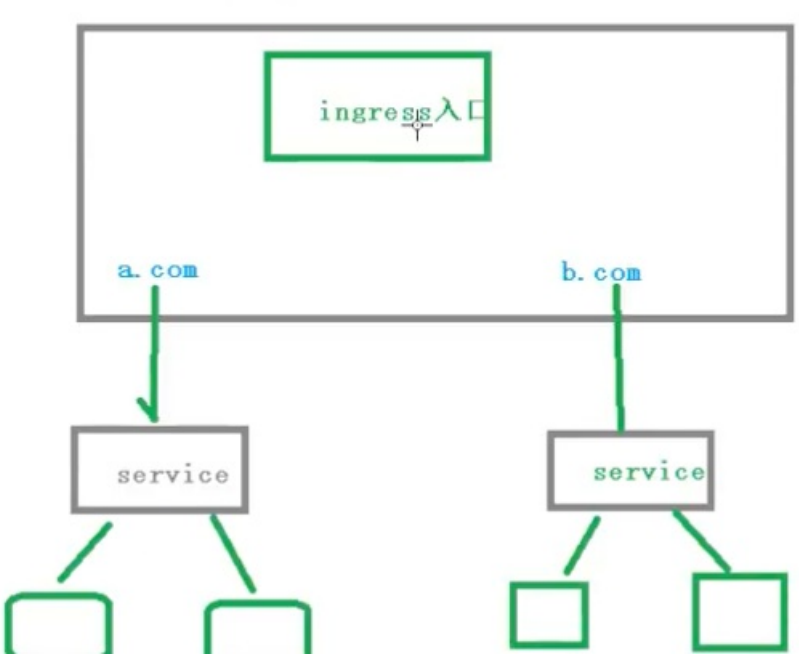
```
[root@m1 mary]# vi rbac-user.sh
[root@m1 mary]# cp /root/TLS/k8s/ca* ./
[root@m1 mary]# bash rbac-user.sh
2020/09/03 17:03:10 [INFO] generate received request
2020/09/03 17:03:10 [INFO] received CSR

[root@m1 mary]# kubectl get pods -n roledemo
```

說明：

1. **命名空間** 用於區分不同的資源和工作負載。
2. **角色和角色綁定** 控制對特定命名空間的訪問權限。
3. **使用證書識別身份** 是基於 RBAC 的身份驗證機制，確保用戶能正確訪問資源。

Ingress 概述



Ingress

1. 將端口號對外暴露，通過 IP + 端口號進行訪問

- **使用 Service 裡面的 NodePort 實現**

---

2. NodePort 缺陷

- **在每個節點上都會起到端口**，在訪問時可以通過任何節點，通過節點 IP + 暴露端口號實現訪問。
- 意味著每個端口只能使用一次，一個端口對應一個應用。
- **實際訪問中都是用域名**，根據不同域名跳轉到不同端口服務中。

---

3. Ingress 和 Pod 的關係d

- **Pod 和 Ingress 是通過 Service 關聯的**。
- **Ingress 作為統一入口**，由 Service 關聯一組 Pod。

說明：

- **Ingress** 提供基於域名的訪問路由能力，實現集群內外通信的靈活控制。
- 它比 NodePort 更靈活且節約端口資源，是 Kubernetes 中常用的負載均衡工具之一。

以下是圖片內容的文字辨識及翻譯成繁體中文：

---

5. 使用 Ingress

- **第一步：** 部署 Ingress Controller
- **第二步：** 創建 Ingress 規則

我們在這裡選擇官方維護的 NGINX 控制器，實現部署。

以下是圖片中的文字辨識及翻譯成繁體中文：

---

6. 使用 Ingress 對外暴露應用

(1) 創建 Nginx 應用，對外暴露端口使用 NodePort

```
kubectl create deployment web --image=nginx

kubectl expose deployment web --port=80 --target-port=80 --type=NodePort
```

---

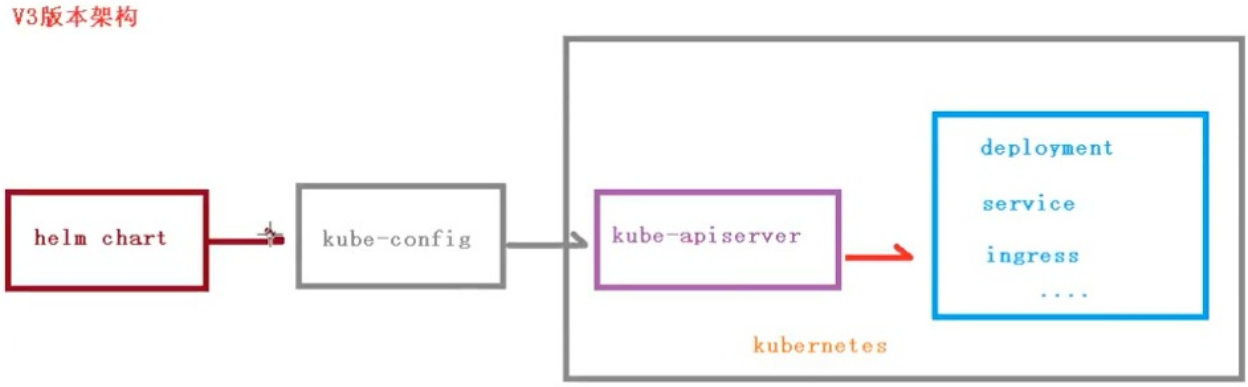
(2) 部署 Ingress Controller

```
[root@k8smaster ~]# vi ingress-con.yaml
[root@k8smaster ~]# kubectl apply -f ingress-con.yaml
namespace/nginx-ingress created
configmap/nginx-configuration created
configmap/tcp-services created
configmap/udp-services created
serviceaccount/nginx-ingress-serviceaccount created
clusterrole.rbac.authorization.k8s.io/nginx-ingress-clusterrole created
role.rbac.authorization.k8s.io/nginx-ingress-role created
rolebinding.rbac.authorization.k8s.io/nginx-ingress-role-nisa-binding created
```

說明：

1. **NodePort 暴露應用：**
   - 通過 `NodePort` 將應用對外暴露，允許外部訪問特定端口。
2. **Ingress Controller 部署：**
   - 使用配置文件 `ingress-con.yaml` 部署 Ingress 控制器，啟用基於域名的路由功能。

helm 概述



Helm

1. Helm 引入

1. **之前方式部署應用的基本過程：**
   - 編寫 YAML 文件：
     - Deployment
     - Service
     - Ingress
2. **如果使用之前方式部署單一應用，少數服務的應用，比較合適：**
   - 例如部署微服務項目，可能有幾十個服務，每個服務都有一套 YAML 文件，需要維護大量 YAML 文件，版本管理特別不方便。

---

2. 使用 Helm 可以解決哪些問題？

1. 使用 Helm 可以把這些 YAML 作為一個整體管理。
2. 實現 YAML 高效複用。
3. 使用 Helm 應用級別的版本管理。

---

3. Helm 介紹

Helm 是一個 Kubernetes 的包管理工具，類似於 Linux 下的包管理器，如 yum/apt 等，可以很方便地將之前打包好的 YAML 文件部署到 Kubernetes 上。

4.Helm 有 3 個重要概念：

1. **Helm**：
   一個命令行客戶端工具，主要用於 Kubernetes 應用 Chart 的創建、打包、發布和管理。
2. **Chart**：
   應用描述，一系列用於描述 Kubernetes 資源相關文件的集合。
3. **Release**：
   基於 Chart 的部署實體。一個 Chart 被 Helm 運行後將會生成對應的一個 Release；
   它會在 Kubernetes 中創建出真實運行的資源對象。

5.Helm 在 2019 年發布 V3 版本，和之前版本相比有變化

1. **V3 版本刪除 Tiller**，架構變化。
2. **Release** 可以在不同命名空間重用。
3. **將 Chart 推送到 Docker 倉庫中**。

以下是圖片內容的文字辨識及翻譯成繁體中文：

1. Helm 安裝

1. **下載 Helm 安裝壓縮文件，上傳到 Linux 系統中**

```
[root@k8smaster ~]# tar zxvf helm-v3.0.0-linux-amd64.tar.gz
linux-amd64/helm
linux-amd64/README.md
linux-amd64/LICENSE
```

2. **解壓 Helm 壓縮文件，將解壓後的 Helm 目錄複製到 `/usr/bin` 目錄下**

```
[root@k8smaster linux-amd64]# mv helm /usr/bin
[root@k8smaster linux-amd64]# helm
```

2. 配置 Helm 倉庫

(1) 添加倉庫

```
helm repo add 倉庫名稱 倉庫地址
```

示例：

```
[root@k8smaster linux-amd64]# helm repo add stable http://mirror.azure.cn/kubernetes/charts
"stable" has been added to your repositories
```

檢查倉庫列表：

```
[root@k8smaster linux-amd64]# helm repo list
NAME    URL
stable  http://mirror.azure.cn/kubernetes/charts
```

(2) 更新倉庫地址

```
[root@k8smaster linux-amd64]# helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "aliyun" chart repository
...Successfully got an update from the "stable" chart repository
Update Complete. Happy Helming!
```

(3) 刪除倉庫

```
[root@k8smaster linux-amd64]# helm repo remove aliyun
"aliyun" has been removed from your repositories
```

說明：

- 使用 `helm repo remove <倉庫名稱>` 可以刪除已添加的 Helm 倉庫。
- 這樣可以清理不再需要的倉庫條目，保持倉庫列表整潔。

以下是圖片內容的文字辨識及翻譯成繁體中文：

```
[root@k8smaster linux-amd64]# helm search repo weave
NAME                CHART VERSION   APP VERSION     DESCRIPTION
```

```
stable/weave-cloud   0.3.7          1.4.0         Weave Cloud is a add-on...
stable/weave-scope   1.1.10         1.12.0        A Helm chart for the We...

[root@k8smaster linux-amd64]# helm install ui stable/weave-scope
NAME: ui
LAST DEPLOYED: Mon Sep 7 14:22:47 2020
NAMESPACE: default
STATUS: deployed

[root@k8smaster linux-amd64]# helm list
NAME    NAMESPACE   REVISION    UPDATED                                 STATUS
ui      default     1           2020-09-07 14:22:47.383427149 +0800 CST deployed

[root@k8smaster linux-amd64]# helm status ui
NAME: ui
LAST DEPLOYED: Mon Sep 7 14:22:47 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
```

---

1. **搜索 Helm 倉庫中的 Weave 應用**
   - 使用命令 `helm search repo weave`，顯示兩個可用的應用：
     - **stable/weave-cloud**：版本 0.3.7，應用版本 1.4.0。
     - **stable/weave-scope**：版本 1.1.10，應用版本 1.12.0。
2. **安裝 Weave Scope**
   - 使用命令 `helm install ui stable/weave-scope` 將 Weave Scope 安裝到 Kubernetes 中，並命名為 **ui**。
3. **查看已安裝的 Helm 應用列表**
   - 使用命令 `helm list`，顯示應用 **ui** 已成功部署，位於命名空間 **default**。
4. **查看應用狀態**
   - 使用命令 `helm status ui`，顯示應用名稱 **ui**，狀態為 **deployed**，部署版本為 **1**。

以下是圖片內容的文字辨識及翻譯成繁體中文：

如何自己創建 Chart

1. 使用命令創建 Chart

```
helm create chart名稱
```

示例：

```
[root@k8smaster ~]# helm create mychart
Creating mychart
[root@k8smaster mychart]# ls
charts  Chart.yaml  templates  values.yaml
```

- **Chart.yaml**：當前 Chart 屬性配置信息。
- **templates**：編寫 YAML 文件放到這個目錄中。
- **values.yaml**：YAML 文件可以使用全局變量。

---

2. 在 templates 文件夾創建兩個 YAML 文件

- **deployment.yaml**
- **service.yaml**

示例：

```
[root@k8smaster templates]# ls
deployment.yaml  service.yaml
```

說明：

1. **Helm create** 命令自動生成 Chart 的目錄結構和必要文件，方便進一步編輯。

2. **templates 文件夾** 是放置 Kubernetes 配置文件（如 Deployment、Service 等 YAML 文件）的地方。
3. **values.yaml** 文件允許定義變量，提供動態和靈活的配置支持。

以下是圖片內容的文字辨識及翻譯成繁體中文：

---

3. 安裝 mychart

```
[root@k8smaster ~]# helm install web1 mychart/
NAME: web1
LAST DEPLOYED: Mon Sep 7 15:03:17 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

說明：

1. **命令解釋**：
   - `helm install`：用於安裝 Chart。
   - `web1`：為此次安裝命名為 `web1`。
   - `mychart/`：指定需要安裝的 Chart 目錄。
2. **輸出信息**：
   - **NAME**：安裝的應用名稱，這裡是 `web1`。
   - **LAST DEPLOYED**：最後部署的時間。
   - **NAMESPACE**：部署的命名空間，默認為 `default`。
   - **STATUS**：應用部署狀態，這裡是 `deployed`（已部署）。
   - **REVISION**：部署的版本號，這裡是 `1`。
   - **TEST SUITE**：測試套件信息，這裡是 `None`（沒有測試）。

---

4. 應用升級

```
helm upgrade chart名稱
```

示例：

```
[root@k8smaster ~]# helm upgrade web1 mychart/
Release "web1" has been upgraded. Happy Helming!
```

說明：

1. **命令解釋**：
   - `helm upgrade`：用於升級已部署的 Chart 應用。
   - `web1`：需要升級的應用名稱。
   - `mychart/`：指定升級所基於的 Chart 目錄。
2. **輸出信息**：
   - **Release "web1" has been upgraded**：表示應用 `web1` 已成功升級。

以下是圖片內容的文字辨識及翻譯成繁體中文：

實現 YAML 高效複用

- **通過傳遞參數，動態渲染模板，YAML 內容動態傳入參數生成。**

```
[root@k8smaster mychart]# ls
charts  Chart.yaml  templates  values.yaml
```

- **在 Chart 中有 `values.yaml` 文件，定義 YAML 文件全局變量。**

操作步驟：

1. **在 `values.yaml` 定義變量和值。**
2. **在具體 YAML 文件中，獲取定義變量值。**

說明：

- YAML 文件大體有幾個地方不同的參數：
  - **image**
  - **tag**
  - **label**
  - **port**
  - **replicas**

說明：

使用 `values.yaml` 提高了 Chart 的靈活性，可以輕鬆實現模板的複用和參數化，方便不同環境配置。

---

k8s 持久化儲存

以下是圖片內容的文字辨識及翻譯成繁體中文：

數據卷 emptydir，是本地存儲，pod 重啟，數據不存在了，需要對數據持久化存儲

1. NFS，網絡存儲

- **pod 重啟，數據還存在的方式**

第一步：找一台服務器作為 NFS 服務端

(1) 安裝 NFS

```
yum install -y nfs-utils
```

---

(2) 設置掛載路徑

```
[root@atonline ntest /]# vi /etc/exports
/data/nfs *(rw,no_root_squash)
```

---

(3) 掛載路徑需要創建出來

```
[root@atonline ntest data]# mkdir nfs
[root@atonline ntest data]# ls
nfs
```

說明：

1. **emptydir** 是 Kubernetes 中的一種臨時存儲方式，pod 重啟後數據會丟失。
2. **NFS（網絡文件系統）** 提供一種數據持久化存儲解決方案，pod 重啟後數據仍然保留。
3. **操作流程**：
   - 安裝 NFS 軟件。
   - 配置 NFS 的掛載路徑。
   - 確保掛載的目錄已創建。

以下是圖片內容的文字辨識及翻譯成繁體中文：

第二步：在 k8s 集群 node 節點安裝 NFS

```
yum install -y nfs-utils
```

說明：

- 在 Kubernetes 集群的每個節點上安裝 **NFS 客戶端工具**（`nfs-utils`），以便節點能夠掛載 NFS 共享存儲。

以下是圖片內容的文字辨識及翻譯成繁體中文：

第三步：在 NFS 服務器啟動 NFS 服務

```
# systemctl start nfs
# ps -ef | grep nfs
15:03 ? 00:00:00 [nfsd4_callbacks]
```

第四步：在 k8s 集群部署應用使用 NFS 持久化網絡存儲

```
vi nfs-nginx.yaml
kubectl apply -f nfs-nginx.yaml

[root@k8smaster pv]# kubectl exec -it nginx-depl-6b96bc8d7d-92qhv bash
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version.

root@nginx-depl-6b96bc8d7d-92qhv:/# ls /usr/share/nginx/html
index.html
```

說明：

第三步：啟動 NFS 服務

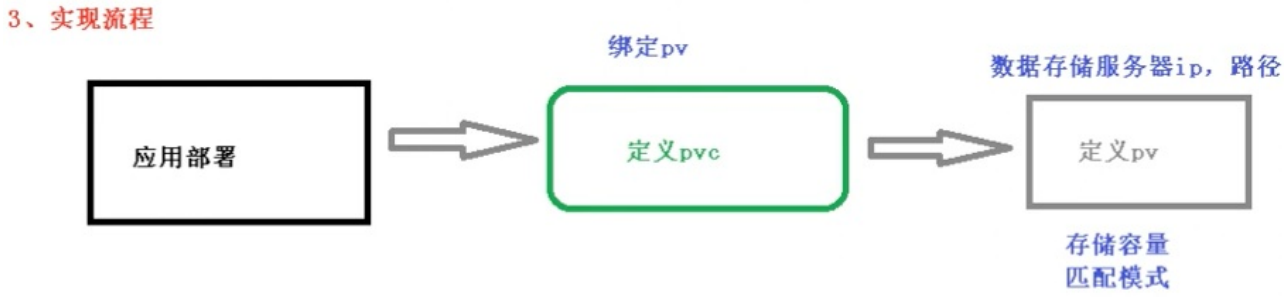
1. **啟動 NFS 服務**：通過 `systemctl start nfs`。
2. **驗證 NFS 啟動狀態**：使用 `ps -ef | grep nfs` 查看進程是否正常啟動。

第四步：部署應用並使用 NFS

1. **編輯 YAML 文件**：創建或修改 `nfs-nginx.yaml` 文件，配置 NFS 為持久化存儲。
2. **應用配置文件**：使用 `kubectl apply` 部署應用。
3. **驗證 NFS 存儲是否生效**：
   - 使用 `kubectl exec` 進入 Pod。
   - 確認存儲目錄（如 `/usr/share/nginx/html`）內的文件是否存在，例如 `index.html`。

PV 和 PVC

1. **PV（Persistent Volume，持久化存儲）**：
   - 持久化存儲，對存儲資源進行抽象。
   - 對外提供可調用的地方（生產者）。
2. **PVC（Persistent Volume Claim，持久化存儲請求）**：
   - 用於調用，不需要關心內部實現細節（消費者）。



k8s 監控

集群資源監控

1. 監控指標

- **集群監控**
  - 節點資源利用率
  - 節點數量
  - 運行的 Pods
- **Pod 監控**
  - 容器指標
  - 應用程序

---

2. 監控平台搭建方案：Prometheus + Grafana

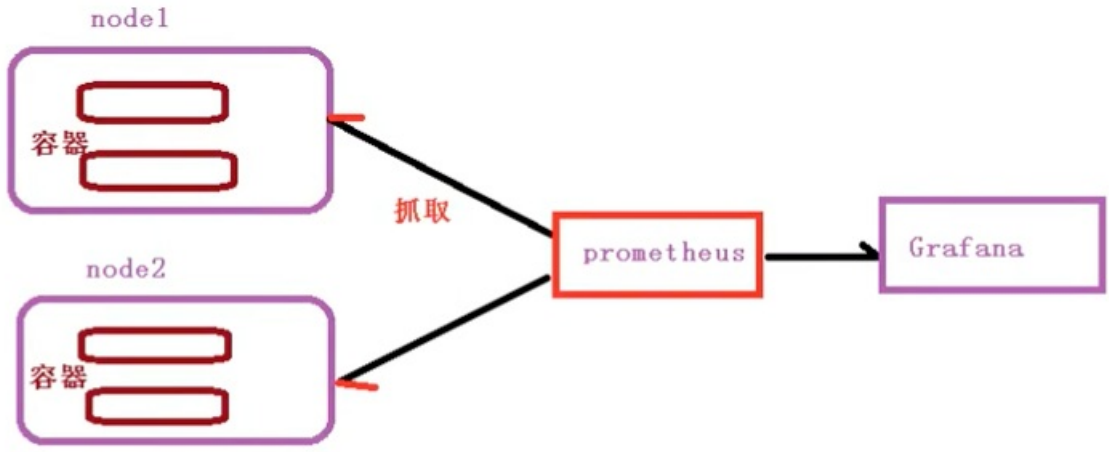
(1) Prometheus

- 開源的
- 具備監控、報警、數據庫功能
- 以 HTTP 協議定期抓取被監控組件的狀態
- 不需要複雜的集成過程，使用 HTTP 接口接入即可

---

(2) Grafana

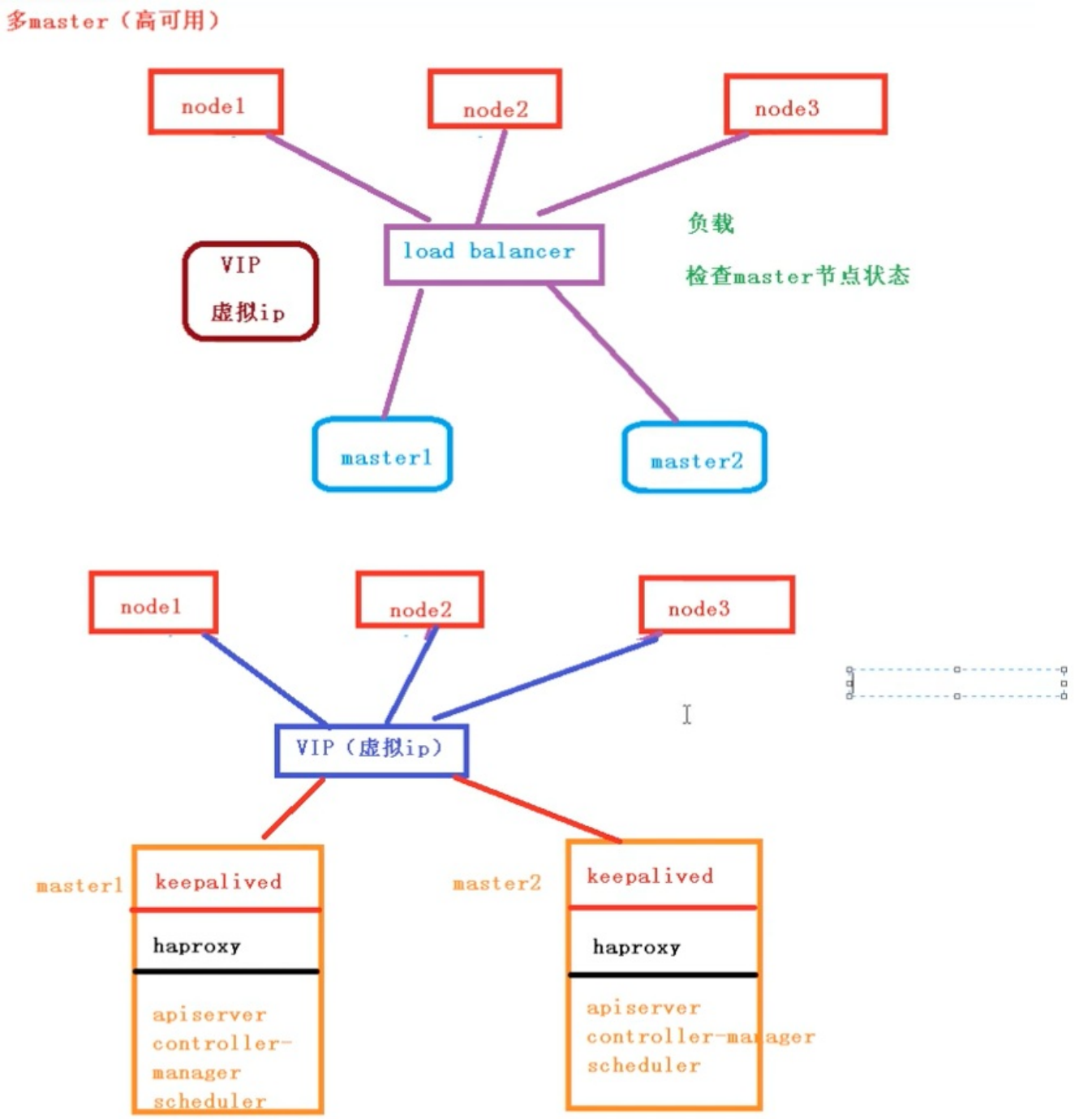
- 開源的數據分析和可視化工具
- 支援多種數據源




```
# rbac-setup.yaml
apiGroups: ["*"]
resources:
  - nodes
  - nodes/proxy
  - services
  - endpoints
  - pods
verbs: ["get", "list", "watch"]
---
apiGroups:
  - extensions
resources:
  - ingresses
verbs: ["get", "list", "watch"]
nonResourceURLs: ["/metrics"]
verbs: ["get"]
---
```

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
  - kind: ServiceAccount
    name: prometheus
    namespace: kube-system
```

k8s 高可用集群



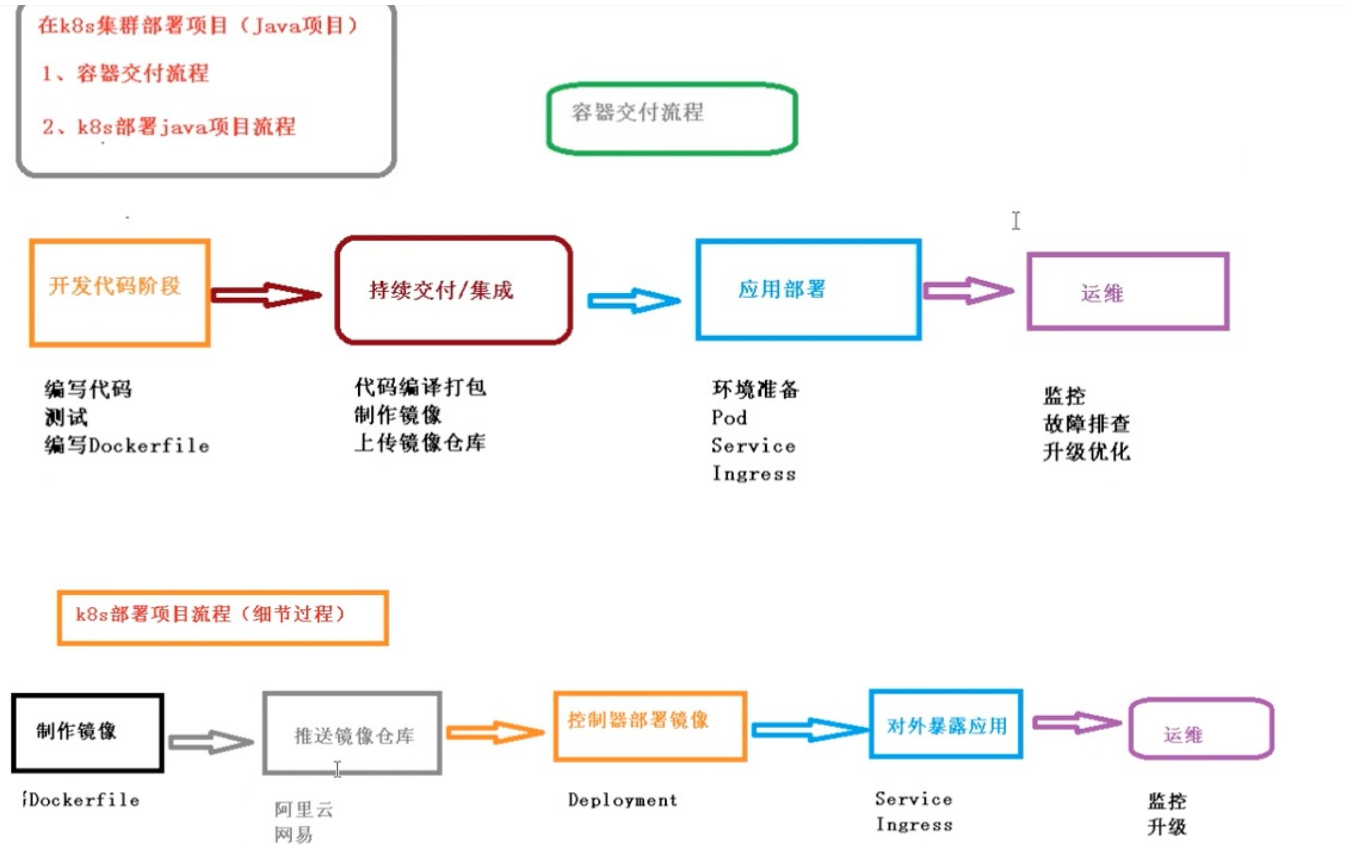
master1	192.168.44.155
master2	192.168.44.156
node1	192.168.44.157
VIP	192.168.44.158

1、部署keepalived 2、部署haproxy 3、初始化操作 4、安装docker，网络插件

1、部署keepalived 2、部署haproxy 3、添加master2节点到集群 安装docker，网络插件

加入到集群中 安装docker，网络插件

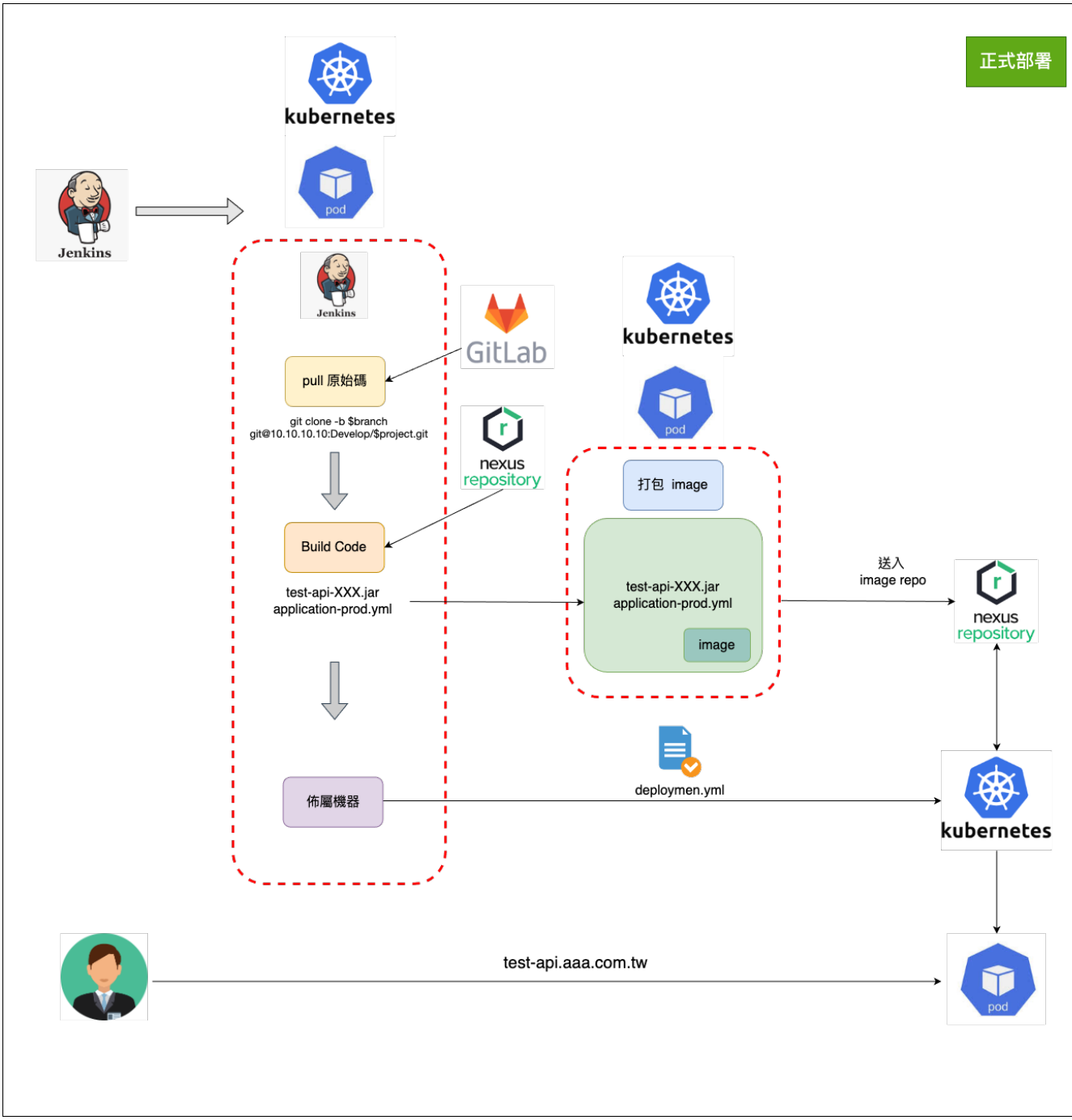
k8s 部署流程



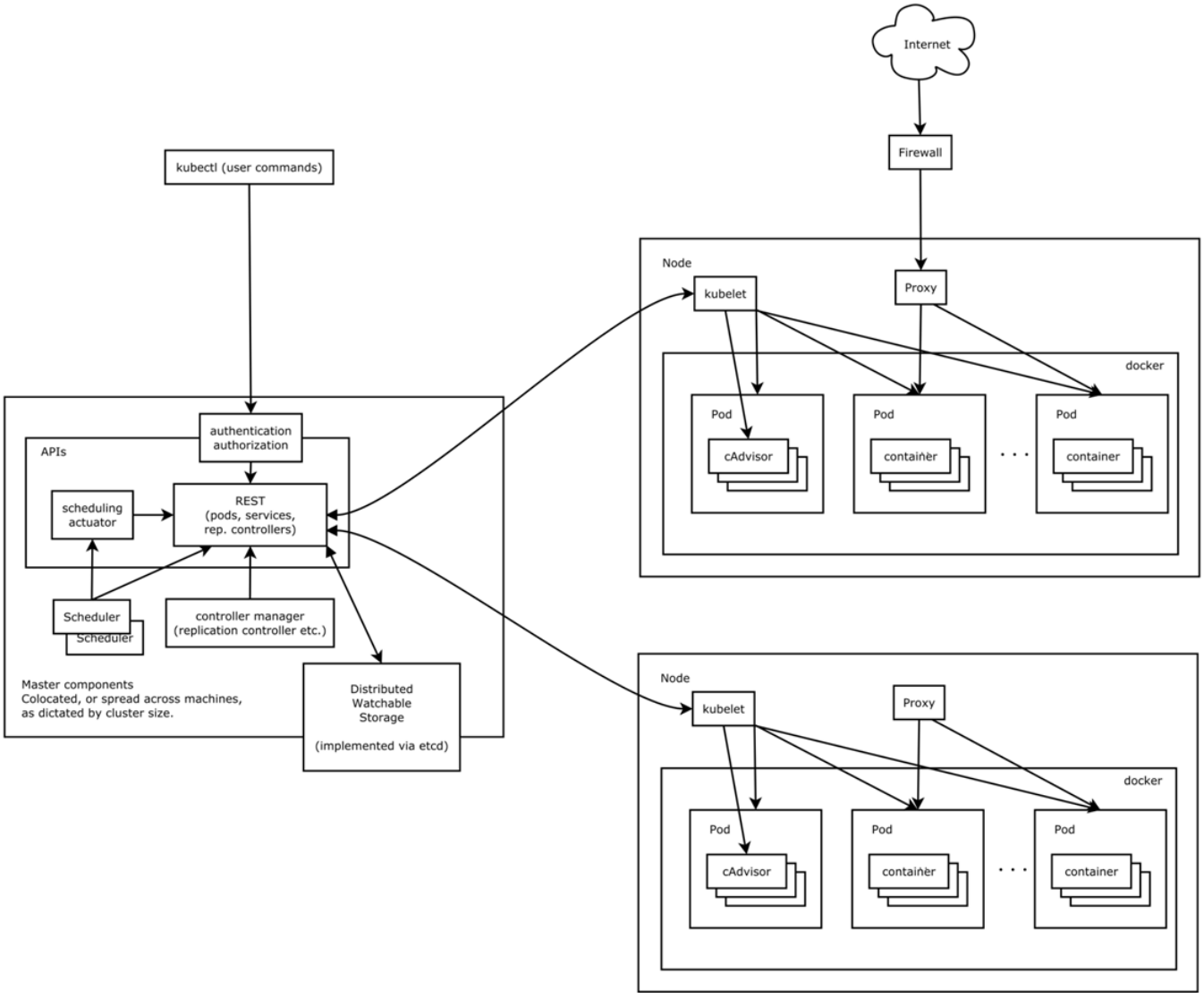

```
FROM ubuntu:22.04

# 設定時區為台北
ENV TZ=Asia/Taipei
RUN apt-get update && apt-get install -y tzdata && \
    ln -sf /usr/share/zoneinfo/$TZ /etc/localtime && \
    echo $TZ > /etc/timezone && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# 複製並執行安裝腳本
COPY install.sh /tmp/install.sh
RUN chmod +x /tmp/install.sh && /tmp/install.sh
```


正式部署
kubernetes
pod
Jenkins
Jenkins
GitLab
pull 原始碼
git clone -b $branch
git@10.10.10.10:Develop/$project.git
nexus
repository
Build Code
test-api-XXX.jar
application-prod.yml
佈屬機器
kubernetes
pod
打包 image
test-api-XXX.jar
application-prod.yml
image
送入
image repo
nexus
repository
deploymen.yml
kubernetes
test-api.aaa.com.tw
pod

k8s核心組件



---

3. k8s 集群架構組件

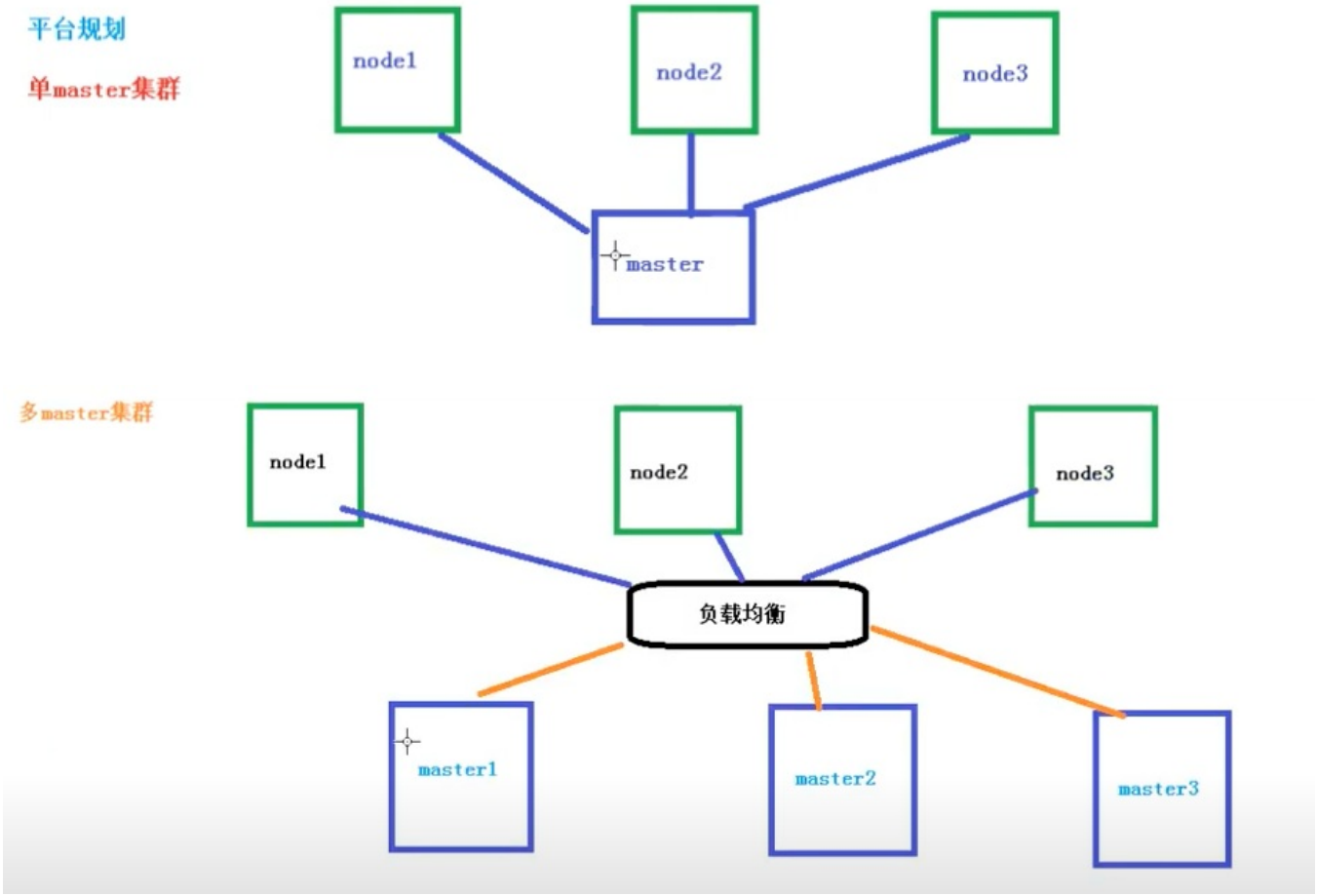
- **Master**（主控節點）和 **Node**（工作節點）

(1) Master 組件

- **apiserver**
  集群統一入口，以 RESTful 方式，交給 etcd 存儲
- **scheduler**
  節點調度，選擇 Node 節點應用部署
- **controller-manager**
  處理集群中常規的後台任務，一個資源對應一個控制器
- **etcd**
  存儲系統，用於保存集群相關的數據

(2) Node 組件

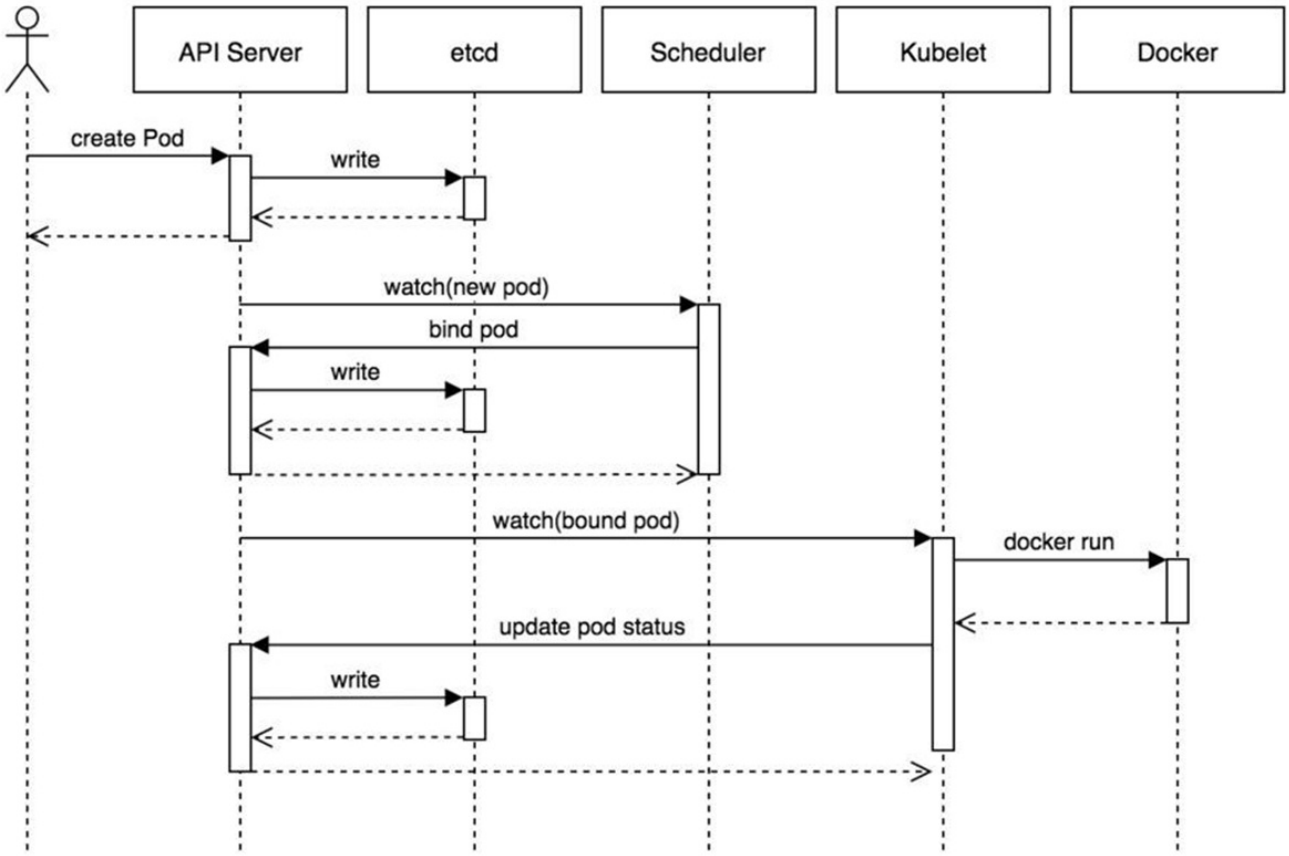
- **kubelet**
  - Master 排到 Node 節點代表，管理本機容器。
- **kube-proxy**
  - 提供網絡代理，負載均衡等操作。



K8S 核心概念

1. **Pod**
   - 最小部署單元
   - 一組容器的集合
   - 共享網絡
   - 生命週期是短暫的
2. **Controller**
   - 確保預期的 Pod 副本數量
   - 無狀態應用部署
   - 有狀態應用部署
   - 確保所有的 Node 運行同一個 Pod
   - 一次性任務和定時任務
3. **Service**
   - 定義一組 Pod 的訪問規則

如有其他需求，請隨時告訴我！



kubectl run podtest --image=nginx

1. kubectl向apiserver發送一個創建pod的請求，apiserver會將數據放到etcd存儲。

2. Scheduler通過list-watch的方式，收到未綁定pod資源，通過自身調度算法選擇一個合適的node進行綁定，然後響應給apiserver，把信息更新到etcd中。

3. Kubelet同樣通過list-watch的方式，收到分配到自己節點上pod，調用docker api創建容器，然後將容器狀態響應給apiserver，同時，把容器的信息、事件及狀態也通過apiserver寫入到etcd中。