



K8s 設定副本數



詳細說明:k8s replica 設成2



在 Kubernetes (K8s) 中，將某個 Deployment 或 ReplicaSet 的 `replicas` 設定為 `2`，表示希望該應用程式的 Pod 數量維持在 2 個，以達到負載均衡與高可用性的目的。

具體操作方式：

1. 修改 Deployment 物件

如果你的應用程式是透過 Deployment 管理的，可以直接修改 `replicas` 數量。

方法 1：使用 `kubectl scale` 指令

你可以使用 `kubectl scale` 命令來調整副本數：

```
kubectl scale deployment <deployment-name> --replicas=2
```

這會將指定的 Deployment 副本數調整為 2。

方法 2：編輯 YAML 並更新

你也可以直接修改 Deployment 的 YAML 定義。

使用以下指令來編輯現有的 Deployment：

```
kubectl edit deployment <deployment-name>
```

找到 `spec.replicas`，並將其值設為 `2`：





```
spec:
  replicas: 2
```

儲存並退出，Kubernetes 會自動調整副本數。

方法 3：使用 `apply` 更新

如果你有 Deployment 的 YAML 檔案，例如 `deployment.yaml`，可以手動修改 `replicas` 欄位：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: my-image:latest
          ports:
            - containerPort: 80
```

然後使用 `kubectl apply` 來更新：

```
kubectl apply -f deployment.yaml
```



2. 修改 ReplicaSet 物件

如果你的應用程式是透過 ReplicaSet 而非 Deployment 管理的，你可以執行：

```
kubectl scale replicaset <replicaset-name> --replicas=2
```

或修改 ReplicaSet 的 YAML：

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: my-image:latest
          ports:
            - containerPort: 80
```

然後使用：

```
kubectl apply -f replicaset.yaml
```

3. 驗證是否生效



你可以使用以下指令確認 Pod 是否正確建立：

```
kubectl get pods
```

應該會看到 2 個相同的 Pod 正在運行。

你也可以查看 Deployment 或 ReplicaSet 的狀態：

```
kubectl get deployment <deployment-name>  
kubectl get replicaset <replicaset-name>
```

這些指令會顯示 `DESIRED`、`CURRENT`、`AVAILABLE` 等欄位，確認副本數量是否正確。

這樣就成功將 K8s 的 `replicas` 設為 `2` 了！如果你的應用程式需要自動調整副本數，也可以考慮使用 **Horizontal Pod Autoscaler (HPA)** 來動態擴展。



詳細說明:k8s Deployment replica 設成2用意



在 Kubernetes (K8s) 中，將 Deployment 的 `replicas` 設為 `2` 的主要目的是提高**可用性 (Availability)** 和**負載分擔 (Load Balancing)**，確保應用程式能夠穩定運行，即使有部分 Pod 發生故障也不會影響服務。

1. 設定 `replicas: 2` 的用意

(1) 高可用性 (High Availability)



當 `replicas: 2`，Kubernetes 會確保始終有 **2 個 Pod** 在運行，即使其中一個 Pod 發生錯誤、當機或被刪除，系統仍然能夠維持正常運作。例如：

- 某個 Pod 崩潰時，另一個 Pod 仍然可用，避免服務中斷。
- 滾動更新 (Rolling Update) 過程中，不會出現短暫的停機情況 (Zero Downtime Deployment)。

舉例：

如果 `replicas: 1`，而該唯一的 Pod 崩潰或節點 (Node) 故障，則服務會停擺。但如果 `replicas: 2`，則至少會有一個 Pod 可用，服務仍能持續運行。

(2) 負載均衡 (Load Balancing)

當 `replicas` 設定為 `2` 時，Kubernetes 的 Service (通常搭配 `type: ClusterIP` 或 `type: LoadBalancer`) 會透過 **內建的 LoadBalancer** 在多個 Pod 之間均勻分配流量，減輕單一 Pod 的負擔，提升整體效能。

示例：

假設一個 Web 應用程式的 `replicas: 2`，且有一個 Service 來負責流量分配：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
```



```
containers:
  - name: my-app
    image: my-image:latest
    ports:
      - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

當有多個使用者訪問 `my-service` 時，Kubernetes 會透過內部的 **Round-Robin** 負載均衡策略，將流量均勻地分配給 2 個 Pod，避免單一 Pod 過載。

(3) 滾動更新 (Rolling Update)

當應用程式需要升級時，Kubernetes **滾動更新** (`RollingUpdate`) 機制可以確保不會影響服務的可用性。例如：

- Kubernetes 會先啟動一個新版本的 Pod，然後再關閉舊版本的 Pod。
- 如果 `replicas: 2`，至少會有 1 個舊版 Pod 和 1 個新版 Pod 同時運行，確保應用不會因為更新而中斷。

```
strategy:
  type: RollingUpdate
  rollingUpdate:
```



```
maxUnavailable: 1  
maxSurge: 1
```

這樣，在更新過程中，**至少有 1 個 Pod 能夠提供服務**，避免整個應用停機。

(4) 抵禦單點故障 (Fault Tolerance)

當 `replicas` 設定為 `2`，Kubernetes 會根據 Node 的可用性自動調度 Pod，避免所有 Pod 都集中在同一個節點上，提升系統的**容錯能力**。

- 若某個 Node 故障，Kubernetes 會自動在其他可用的 Node 上重新啟動 Pod，確保服務不中斷。
- 若 **Pod 只設定 1 個** (`replicas: 1`)，當該 Pod 崩潰或所在的 Node 當機時，應用程式會完全無法使用。

示例：

假設你有 3 個 Kubernetes 節點：

```
Node-1    [Pod-1]  
Node-2    [Pod-2]  
Node-3    [空]
```

如果 `Node-1` 故障，`Pod-1` 會被 Kubernetes 自動重新調度到 `Node-3`，確保 `Pod-2` 仍然在運行，避免整個應用程式停擺。

(5) 支援自動擴展 (Auto Scaling)

當 `replicas: 2`，你可以進一步搭配 **水平自動擴展 (HPA, Horizontal Pod Autoscaler)** 來根據流量負載動態調整 Pod 數量。例如：



```
kubectl autoscale deployment my-app --cpu-percent=50 --min=2 --max=5
```

這樣當 CPU 使用率超過 50% 時，Pod 數量會自動擴展，從 `2` 增加到 `5`，提高彈性伸縮能力。

2. `replicas: 2` VS `replicas: 1` VS `replicas: N`

Replicas	優點	缺點	適用場景
`1`	節省資源、適用於開發環境	無法提供高可用性、無負載均衡	測試環境、本地開發
`2`	提供基本的高可用性、負載均衡	如果一個 Pod 故障，仍然只有 1 個可用	小型生產環境
`N` (如 3, 5, 10)	更好的可用性、負載均衡、更適合擴展	佔用更多資源	大型生產環境

如果你的應用程式是**小型 Web 服務**或**企業內部系統**，`replicas: 2` 是一個合理的數值，既能提供高可用性，又不會浪費過多資源。

3. 結論

將 K8s Deployment 的 `replicas` 設為 `2` 主要是為了： **確保應用高可用性** (避免單點故障)

 **實現負載均衡** (分散流量，減輕單一 Pod 負擔)

 **支援滾動更新** (應用升級時不中斷服務)

 **提升容錯能力** (Pod 故障時仍有可用實例)

 **適應自動擴展** (可根據負載動態增加 Pod)



因此，`replicas: 2` 是 Kubernetes **最小可行的高可用性設定**，適合中小型應用程式的生產環境。如果流量需求更高，可以適當提升副本數量（`replicas: 3, 5, 10...`）。



```
這個exportery在k8s Deployment replica 設成2，import
csv
import os
import time
import logging
from datetime import datetime
from prometheus_client.core import
GaugeMetricFamily, REGISTRY
from prometheus_client.exposition import
MetricsHandler
from http.server import HTTPServer
from socketserver import ThreadingMixIn
from logging.handlers import RotatingFileHandler
from threading import Lock

# 設置日誌輪替
log_handler = RotatingFileHandler(
    "exporter.log", maxBytes=5 * 1024 * 1024,
    backupCount=3
)
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(
message)s",
    handlers=[log_handler]
)

class LogExporter:
    def __init__(self, log_file):
        self.log_file = log_file
        self.tmp_log_file = None
        self.metric_cache = {}
        self.lock_file = f"{log_file}.lock"
        self.cache_lock = Lock()
        self.update_timestamp = None
```



```
self.scrapers_access_record = {} # 記錄 Scraper 是否已抓取

def collect(self, scraper_id):
    """確保 Scraper 在 metric 更新週期內只能抓取一次"""

    with self.cache_lock:
        # 如果 Scraper 已經抓取過這個 metric 週期，則拒絕
        if scraper_id in self.scrapers_access_record:
            logging.warning(f"Scraper {scraper_id} already accessed metrics in this cycle.")
            return

        # 記錄 Scraper 這次抓取的時間
        self.scrapers_access_record[scraper_id] = time.time()

        metric = GaugeMetricFamily(
            "log_host_job_count",
            "Count of occurrences of host and job_name in log",
            labels=["host", "job_name"]
        )
        for (host, job_name), count in self.metric_cache.items():
            metric.add_metric([host, job_name], count)

        yield metric # 返回 metric 指標

def update_metrics(self):
    """從最新的 tmp_log_<timestamp>.csv 更新 metric"""
    if not self.tmp_log_file or not os.path.exists(self.tmp_log_file):
        logging.warning(f"Temporary log file {self.tmp_log_file} does not exist.")
        return

    counts = self._count_host_job(self.tmp_log_file)
```



```
with self.cache_lock:
    self.metric_cache = counts
    self.update_timestamp = time.time()
    self.scrapers_access_record.clear() # 清空
    Scrapper 記錄，允許 Scrapper 再次抓取

logging.info("Metrics updated successfully.")

def _count_host_job(self, tmp_log_file):
    """計算 tmp_log_<timestamp>.csv 中 host 和
    job_name 的出現次數"""
    counts = {}
    try:
        with open(tmp_log_file, 'r') as f:
            reader = csv.reader(f)
            for row in reader:
                host, job_name = row[0], row[1]
                key = (host, job_name)
                counts[key] = counts.get(key, 0) + 1
    except Exception as e:
        logging.error(f"Error reading log file
        {tmp_log_file}: {e}")
    return counts

# 自訂 HTTP Handler 來處理 User-Agent
class CustomMetricsHandler(MetricsHandler):
    def do_GET(self):
        """攔截 Scrapper 的請求，獲取 User-Agent"""
        scraper_id = self.headers.get("User-Agent",
        "unknown") # 取得 Scrapper 的 User-Agent
        logging.info(f"Scrapper {scraper_id} is requesting
        metrics.")

        # 調用 LogExporter 的 collect(), 傳入 Scrapper ID
        for metric in exporter.collect(scraper_id):
            self.wfile.write(metric.serialize())

# 讓 HTTP 伺服器支援多線程，避免 Scrapper 互相影響
```



```
class ThreadingSimpleServer(ThreadingMixIn,
HTTPServer):
    """支援多線程的 HTTP Server"""

if __name__ == "__main__":
    log_file = "log.csv"
    exporter = LogExporter(log_file)

    # 註冊 Prometheus 指標
    REGISTRY.register(exporter)

    # 啟動自訂 HTTP Server (取代 start_http_server())
    server = ThreadingSimpleServer(("0.0.0.0", 8000),
CustomMetricsHandler)
    logging.info("Prometheus exporter running on
http://localhost:8000/metrics")

    # 監控迴圈
    while True:
        # 更新指標快取
        exporter.update_metrics()

        # 等待 Prometheus 抓取指標
        time.sleep(10)
```



這個 `exporter` 是一個基於 **Prometheus Exporter** 的 Python 應用程式，負責解析 `log.csv` 並提供指標 (`metrics`) 給 Prometheus 抓取。現在你想要將它部署到 **Kubernetes**，並設定 **Deployment** 的 `replicas` 為 **2**。這樣的部署方式有幾個關鍵點需要考慮：

1. 為何在 K8s Deployment 設定 `replicas: 2`





在 Kubernetes `Deployment` 中將 `replicas` 設為 `2`，即讓這個 `exporter` 同時運行兩個實例 (Pod)，主要有以下考量：

(1) 高可用性

- 如果某個 `Pod` 當機、節點故障，另一個 `Pod` 仍然可以提供指標，確保 **Prometheus** 仍能夠獲取數據。
- 減少單點故障的影響。

(2) 水平擴展

- 當 **Prometheus** 或 **Scraper** 並發請求增多時，可以透過多個 `Pod` 來分擔請求，提升服務吞吐量。

(3) 避免單點資源瓶頸

- 如果單一 `Pod` 運行 `exporter`，所有的 Scraper 請求都會打到這個 `Pod`，可能導致資源消耗過高。
- 設定 `replicas: 2` 讓 K8s 自動負載均衡，將 Scraper 請求分配給不同的 `Pod`。

(4) 兼容 Kubernetes Service

- 在 `replicas: 2` 的情況下，Kubernetes 內建的 **Service** 會自動使用 `Round-Robin` 方式 負載均衡 Prometheus 的請求，讓多個 `Pod` 共享流量。

2. Kubernetes `Deployment` 設定

我們需要創建一個 **Kubernetes Deployment YAML**，並且確保：

1. 容器運行 **Python** 應用程式
2. 掛載 **Log** 檔案
3. 開放 **Prometheus** 監控的 `/metrics` 端點



4. 允許 Prometheus 透過 Service 訪問

`exporter-deployment.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: log-exporter
  labels:
    app: log-exporter
spec:
  replicas: 2 # 設定 replicas 為 2
  selector:
    matchLabels:
      app: log-exporter
  template:
    metadata:
      labels:
        app: log-exporter
    spec:
      containers:
        - name: log-exporter
          image: my-registry/log-exporter:latest # 你需要事先建立這個 Docker
          ports:
            - containerPort: 8000
          volumeMounts:
            - name: log-storage
              mountPath: /app/logs # 將 log.csv 掛載到這個目錄
      volumes:
        - name: log-storage
          persistentVolumeClaim:
            claimName: log-pvc # 這需要有一個 PVC 來存儲 log.csv
---
apiVersion: v1
kind: Service
metadata:
  name: log-exporter-service
spec:
  selector:
```



```
    app: log-exporter
  ports:
    - protocol: TCP
      port: 8000
      targetPort: 8000
  type: ClusterIP # 內部 Service, 供 Prometheus 抓取
```

3. 確保 `log.csv` 在多個 `Pod` 之間共享

因為 `replicas: 2`，會有 2 個 `Pod` 讀取 `log.csv`，所以我們必須確保：

1. 所有 **Pod** 讀取同一個 `log.csv`，否則不同 Pod 可能讀取不同的日誌，導致 Prometheus 指標不一致。
2. 使用 **Persistent Volume (PV)** 和 **Persistent Volume Claim (PVC)**，讓 `log.csv` 被多個 `Pod` 共享。

`log-pvc.yaml`

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: log-pvc
spec:
  accessModes:
    - ReadWriteMany # 允許多個 Pod 讀取
  resources:
    requests:
      storage: 1Gi
```

這樣，所有 `exporter` Pod 都能夠讀取相同的 `log.csv`。



4. 確保 Prometheus 可以監控這個 `exporter`

Prometheus 需要能夠從 `log-exporter-service` 取得 `/metrics`，所以需要在 Prometheus 的 `scrape_configs` 中加入：

Prometheus 設定

```
scrape_configs:
  - job_name: "log_exporter"
    metrics_path: "/metrics"
    static_configs:
      - targets: ["log-exporter-service:8000"]
```

這樣 Prometheus 會透過 `log-exporter-service` 負載均衡地獲取多個 `Pod` 提供的指標。

5. Docker 映像與 Kubernetes 部署步驟

(1) 建立 Docker 映像

先將 `exporter.py` 轉為 Docker 容器：

```
# Dockerfile
FROM python:3.8
WORKDIR /app
COPY exporter.py .
RUN pip install prometheus_client
CMD ["python", "exporter.py"]
```

然後：

```
docker build -t my-registry/log-exporter:latest .
docker push my-registry/log-exporter:latest # 推送到私有或公有映像倉庫
```




(2) 部署到 Kubernetes

```
kubectl apply -f log-pvc.yaml  
kubectl apply -f exporter-deployment.yaml
```

6. 測試與驗證

(1) 確認 `Pod` 是否運行

```
kubectl get pods
```

應該會看到 `2` 個 `log-exporter` 的 `Pod`。

(2) 測試 Prometheus 指標

```
kubectl port-forward svc/log-exporter-service 8000:8000
```

然後在瀏覽器輸入：

```
http://localhost:8000/metrics
```

應該可以看到 Prometheus 指標，例如：

```
# HELP log_host_job_count Count of occurrences of host and job_name in log  
# TYPE log_host_job_count gauge  
log_host_job_count{host="server1",job_name="backup"} 5.0  
log_host_job_count{host="server2",job_name="update"} 3.0
```

(3) 確保 Prometheus 正確抓取





執行：

```
kubectl logs <log-exporter-pod-name>
```

應該會看到：

```
INFO - Scraper Prometheus is requesting metrics.  
INFO - Metrics updated successfully.
```

7. 總結

- ✓ **K8s Deployment** `replicas: 2` 提供高可用性與負載均衡
- ✓ 透過 **PVC** 確保 `log.csv` 共享，避免不同 Pod 讀取不同的日誌
- ✓ 使用 **Prometheus** `scrape_configs` 配置監控
- ✓ 使用 **Kubernetes Service** 確保 **Prometheus** 抓取時不受 **Pod IP** 變動影響

這樣就成功讓 `exporter` 在 Kubernetes 中以 **多個副本運行**，並讓 Prometheus 能夠正確監控！🎯



k8s Deployment replicas 設成2會造成此exporter metric計算錯誤嗎：



將 Kubernetes **Deployment** 的 `replicas` 設定為 `2` 可能會導致 `exporter` **metric 計算錯誤**，具體影響取決於應用程式如何處理 `log.csv` 和 **Prometheus** 抓取指標的方式。以下是詳細分析：



1. `replicas: 2` 可能導致的 metric 計算問題

如果 `exporter` 的兩個 `Pod` 同時運行並獨立讀取 `log.csv`，可能會導致以下問題：

(1) 重複計算

- 兩個 `Pod` 會各自讀取 `log.csv`，並分別計算 `host` 和 `job_name` 的出現次數，這樣 Prometheus 可能會 **兩次計算相同的數據**，導致數值比實際的還要高。

 **問題示例** 假設 `log.csv` 的內容如下：

```
server1, backup
server2, update
server1, backup
```

當 `replicas: 1` 時，`exporter` 讀取並計算：

```
log_host_job_count{host="server1", job_name="backup"} = 2
log_host_job_count{host="server2", job_name="update"} = 1
```

當 `replicas: 2` 時，每個 `Pod` 都會讀取一次 `log.csv`，導致：

```
log_host_job_count{host="server1", job_name="backup"} = 4 # 本來應該是 2
log_host_job_count{host="server2", job_name="update"} = 2 # 本來應該是 1
```

這會導致 Prometheus **錯誤地記錄指標數據**。

(2) Scraper 記錄不一致

你的 `exporter` 使用 **Scraper 記錄機制** (`self.scrapers_access_record`) 來確保每個 Scraper 在一個週期內只抓取一次 metric。但如果有兩個 `Pod`，Scraper 可能：

- 隨機訪問不同 `Pod`，導致 metric 數據不一致。
- 可能從不同的 `Pod` 獲取不同的數據集，影響 Prometheus 記錄的連續性。





2. 如何解決 metric 計算錯誤？

為了讓 `replicas: 2` 能夠正確運作，可以採取以下策略：

✅ 解法 1：使用 Persistent Volume (PV) 讓所有 Pod 共享 `log.csv`

確保 `log.csv` 被所有 Pod 共享，這樣不同 Pod 讀取的日誌內容是一致的，而不會因為 `log.csv` 存在於不同的 Pod 本地而導致數據不一致。

如何實作？

- 在 `Deployment` 中使用 **Persistent Volume Claim (PVC)** 掛載 `log.csv`。
- 這樣不同 Pod 讀取的 `log.csv` 是同一份文件，確保計算一致。

✅ 解法 2：使用 `sum()` 聚合 Prometheus 指標

如果無法確保 `log.csv` 共享，可以在 Prometheus 設定中 使用 `sum()` 聚合所有 `exporter` Pod 回傳的指標數據：

```
sum(log_host_job_count) by (host, job_name)
```

這樣 Prometheus 會將不同 `Pod` 報告的數據加總，避免數據重複計算的問題。

✅ 解法 3：改變 `exporter` 計算方式

目前 `exporter` 每次都重新讀取 `log.csv` 並計算 **累積值**，這可能導致 `replicas: 2` 時計算重複。



可以改成：

1. 讓 `exporter` 只計算 `log.csv` 最新新增的行，而不是整個檔案重讀。
2. 使用中央 **Redis / Database** 存儲結果，讓所有 `Pod` 共享同一個狀態，而不是獨立計算。

3. 最佳方案

方案	優點	缺點
✅ 使用 PVC 共享 `log.csv`	保持所有 Pod 讀取相同數據，避免重複計算	需要 `ReadWriteMany` 支援
✅ Prometheus 使用 `sum()` 聚合	簡單易行，不需要修改 `exporter`	可能仍然有短暫的數據不一致
✅ 改變 `exporter` 計算方式	避免重複計算，精準追蹤新日誌	需要修改程式邏輯
❌ 直接使用 `replicas: 1`	最簡單，不會計算錯誤	無法提供高可用性，單點故障風險高

4. 結論

如果 `exporter` 直接讀取 `log.csv` 並計算 **累積值**，當 `replicas: 2` 時：

- 可能會 **重複計算**，導致 Prometheus 數據偏高。
- Scraper 可能從不同 `Pod` 抓取不同數據，導致指標不一致。

解決方案

1. 最佳做法：使用 **Persistent Volume (PV/PVC)**，確保所有 Pod 讀取同一份 `log.csv`。



2. **輔助方案**：在 Prometheus 內部 使用 ``sum()`` 聚合，避免數據翻倍。
3. **進階方案**：改變 ``exporter`` 只處理新日誌行，或者 使用 **Redis/DB** 記錄狀態，確保指標計算一致。

👉 如果你能使用 **PVC**，那就是最推薦的方法！如果不能，至少應該讓 **Prometheus** 使用 ``sum()`` 來避免錯誤數據。