



# 使用vmagent替代Prometheus



使用vmagent 代替Prometheus ...  
PDF

參考"使用vmagent 代替Prometheus 採集監控指標- 技術顏良- 博客園.pdf"，以繁體中文詳細說明



## 使用 vmagent 代替 Prometheus 採集監控指標 詳細說明

**vmagent** 是一款能夠替代 **Prometheus** 進行監控指標採集的工具，它能夠從多種來源收集指標，並將這些指標存儲到 **VictoriaMetrics (VM)** 或任何支援 **Prometheus Remote Write** 協議的存儲系統中。

## 1. vmagent 介紹與特性

相比於 **Prometheus**，vmagent 具有更靈活的監控指標採集方式，例如：

### 1. 支援 Pull（拉取）與 Push（推送）

- **Prometheus** 主要採用 **pull 模式**（主動抓取指標）。
- **vmagent** 除了可以 **pull** 也可以 **push**，適用於動態環境。

### 2. 替代 Prometheus 直接作為 scraping target

- vmagent 可以作為一個 **scrape target**，負責收集來自不同來源的數據，再統一發送到遠端存儲。

### 3. 支援 Kafka 讀寫數據

- 可以從 **Kafka** 讀取監控數據，並將數據寫入 **Kafka**。





#### 4. 支援 Prometheus relabeling

- 可以在數據發送到遠端存儲前，**修改、刪除或新增 labels**，提高數據處理靈活性。

#### 5. 支援多種數據協議

- **InfluxDB line**
- **Graphite**
- **OpenTSDB**
- **Prometheus Remote Write**
- **JSON Lines**
- **CSV**

#### 6. 多種存儲系統支援

- 支援數據存儲到 **多個遠端存儲系統**，提高數據冗餘與可靠性。

#### 7. 緩衝機制

- 若遠端存儲不可用，vmagent 會將指標緩存至 `-remoteWrite.tmpDataPath``，當遠端存儲恢復後，會自動將數據發送過去。

#### 8. 降低資源消耗

- 相較於 **Prometheus**，vmagent 使用更少的 **CPU、記憶體、磁碟 I/O 和網路頻寬**，特別適合大規模監控場景。

#### 9. 支援高並發與分佈式架構

- vmagent 可以部署多個實例，將監控目標分配到不同的 **vmagent** 實例，以減少單點壓力。

---

## 2. 在 Kubernetes 環境中部署 vmagent



## 2.1. 設定 RBAC 權限

vmagent 需要從 Kubernetes API Server 獲取資源資訊，因此需要設定 **RBAC (Role-Based Access Control)** 權限。

建立 **RBAC 資源清單 (vmagent-rbac.yaml)**：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: vmagent
  namespace: kube-vm
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: vmagent
rules:
  - apiGroups: ["", "networking.k8s.io", "extensions"]
    resources:
      - nodes
      - nodes/metrics
      - services
      - endpoints
      - endpointslices
      - pods
      - app
      - ingresses
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources:
      - namespaces
      - configmaps
    verbs: ["get"]
  - nonResourceURLs: ["/metrics", "/metrics/resources"]
    verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
```



```
name: vmagent
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: vmagent
subjects:
- kind: ServiceAccount
  name: vmagent
  namespace: kube-vm
```

## 2.2. 設定 vmagent 配置

使用 **ConfigMap** 來管理 **scrape** 設定，如下：

建立 **vmagent 配置 (vmagent-config.yaml)**：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: vmagent-config
  namespace: kube-vm
data:
  scrape.yml: |
    global:
      scrape_interval: 15s
      scrape_timeout: 15s
    scrape_configs:
      - job_name: nodes
        kubernetes_sd_configs:
          - role: node
        relabel_configs:
          - source_labels: [__address__]
            regex: "(.*):10250"
            replacement: "${1}:9111"
            target_label: __address__
            action: replace
```



```
- action: labelmap
  regex: __meta_kubernetes_node_label_(.+)
```

這裡的 **kubernetes\_sd\_configs** 設定讓 **vmagent** 可以透過 Kubernetes API 自動發現 **node** 角色的目標。

## 2.3. 部署 vmagent

建立 **vmagent** 部署資源 (**vmagent-deploy.yaml**)：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vmagent
  namespace: kube-vm
  labels:
    app: vmagent
spec:
  selector:
    matchLabels:
      app: vmagent
  template:
    metadata:
      labels:
        app: vmagent
    spec:
      serviceAccountName: vmagent
      containers:
        - name: agent
          image: "victoriametrics/vmagent:v1.77.0"
          imagePullPolicy: IfNotPresent
          args:
            - -promscrape.config=/config/scrape.yml
            - -remoteWrite.tmpDataPath=/tmpData
            - -remoteWrite.url=http://vminsert:8480/insert/0/prometheus
            - -envflag.enable=true
```



```
- -envflag.prefix=VM_
- -loggerFormat=json
ports:
  - name: http
    containerPort: 8429
volumeMounts:
  - name: tmpdata
    mountPath: /tmpData
  - name: config
    mountPath: /config
volumes:
  - name: tmpdata
    emptyDir: {}
  - name: config
    configMap:
      name: vmagent-config
```

## 3. 使用 vmagent 集群模式

當需要監控大量目標時，可以部署多個 **vmagent** 實例，每個實例負責部分監控目標。

### 3.1. 集群配置

```
args:
  - -promscrape.cluster.membersCount=3
  - -promscrape.cluster.memberNum=0
  - -remoteWrite.url=http://vminsert:8480/insert/0/prometheus
```

- ``-promscrape.cluster.membersCount=3``：定義 **vmagent** 集群共有 3 個實例。
- ``-promscrape.cluster.memberNum=0``：當前 **vmagent** 實例的 ID 為 0。



## 4. 刷新 vmagent 配置

在 Kubernetes 環境下，有兩種方式可以刷新 **vmagent** 配置：

### 1. 發送 SIGHUP 訊號

```
kill -SIGHUP <vmagent-pid>
```

### 2. 發送 HTTP 請求

```
curl -X POST http://vmagent:8429/-/reload
```

## 5. 監控 vmagent 自身

vmagent 提供自身的監控指標：

- `http://vmagent:8429/metrics`：可透過 Prometheus 或 Grafana 監控 **vmagent** 狀態。

## 總結

- vmagent 可替代 **Prometheus**，減少資源消耗。
- 支援 **Pull + Push**，並可與 Kafka、InfluxDB 等協議整合。
- 在 **Kubernetes** 環境，可搭配 **ConfigMap + StatefulSet** 部署。
- 透過 **集群模式** 提高擴展能力。

這樣的架構特別適合 **大規模監控**，同時降低系統資源消耗！🚀



我想要from vmagent的image寫自己的dockerfile，promscrape.cluster.memberNum怎麼使用變數



如果你想要自訂 `vmagent` 的 Dockerfile，並且讓 `-promscrape.cluster.memberNum` 參數可以使用環境變數，這可以透過 **entrypoint script** 或 **Docker ARG/ENV** 來實現。以下是完整的解法：

## 1. 自訂 Dockerfile

在 Dockerfile 中，你可以這樣設定：

```
FROM victoriametrics/vmagent:v1.77.0

# 設定工作目錄
WORKDIR /vmagent

# 複製自訂的配置檔案（可選）
COPY scrape.yml /vmagent/scrape.yml

# 設定環境變數（如果沒指定，則預設為0）
ENV VMAGENT_MEMBER_NUM 0
ENV VMAGENT_CLUSTER_SIZE 3

# 透過 entrypoint script 來處理變數
COPY entrypoint.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh

ENTRYPOINT ["/entrypoint.sh"]
```





## 2. 編寫 `entrypoint.sh` 來動態設定 `memberNum`

在 `entrypoint.sh` 中，使用 **環境變數** 來指定 `promscrape.cluster.memberNum`：

```
#!/bin/sh

# 設定 cluster memberNum，使用環境變數，如果沒設定則預設為0
VMAGENT_MEMBER_NUM=${VMAGENT_MEMBER_NUM:-0}
VMAGENT_CLUSTER_SIZE=${VMAGENT_CLUSTER_SIZE:-3}

exec /vmagent-prod \
  -promscrape.config=/vmagent/scrape.yml \
  -remoteWrite.url=http://vminsert:8480/insert/0/prometheus \
  -promscrape.cluster.membersCount=${VMAGENT_CLUSTER_SIZE} \
  -promscrape.cluster.memberNum=${VMAGENT_MEMBER_NUM} \
  "$@"
```

- `VMAGENT\_MEMBER\_NUM` 預設為 `0`，可以透過 **環境變數** 來覆寫。
- `VMAGENT\_CLUSTER\_SIZE` 預設為 `3`，同樣可以調整。
- `exec` 用來確保 `vmagent` 正常運行，並允許外部參數。

## 3. 構建與執行 Docker 容器

### 3.1. 構建映像

```
docker build -t my-vmagent .
```

### 3.2. 使用環境變數啟動容器

```
docker run -d \
  -e VMAGENT_MEMBER_NUM=1 \
```



```
-e VMAGENT_CLUSTER_SIZE=5 \  
--name my-vmagent my-vmagent
```

這樣，`-promscrape.cluster.memberNum` 會動態設為 **1**，`-promscrape.cluster.membersCount` 設為 **5**。

## 4. 在 Kubernetes 中動態設定

如果你要在 **Kubernetes** 中使用 `StatefulSet` 部署，可以透過 **環境變數** 和 **Pod 名稱** 來自動分配 `memberNum`。

### 4.1. Kubernetes Deployment (StatefulSet)

```
apiVersion: apps/v1  
kind: StatefulSet  
metadata:  
  name: vmagent  
  namespace: kube-vm  
spec:  
  replicas: 3  
  serviceName: vmagent  
  selector:  
    matchLabels:  
      app: vmagent  
  template:  
    metadata:  
      labels:  
        app: vmagent  
    spec:  
      containers:  
        - name: vmagent  
          image: my-vmagent  
          env:  
            - name: VMAGENT_MEMBER_NUM  
              valueFrom:
```



```
fieldRef:
  fieldPath: metadata.name
- name: VMAGENT_CLUSTER_SIZE
  value: "3"
```

這裡使用 `metadata.name` 來動態設定 **VMAGENT\_MEMBER\_NUM**，例如：

- `vmagent-0` → `memberNum=0`
- `vmagent-1` → `memberNum=1`
- `vmagent-2` → `memberNum=2`

## 總結

1. 使用 **Dockerfile** 搭配 **entrypoint.sh** 來讓 `promsrape.cluster.memberNum` 支援環境變數。
2. 使用 `docker run -e` 來設定不同的 `memberNum`。
3. 在 **Kubernetes StatefulSet** 中動態設定 `memberNum` 透過 Pod 名稱來確保正確的 `memberNum` 分配。

這樣可以讓 `vmagent` 在 **動態環境** 中更加靈活 🚀！



詳細說明:entrypoint.sh，有3個pod，這樣會是membernum 多少呢



### 1. `entrypoint.sh` 詳細解析

在 `entrypoint.sh` 中，我們會根據環境變數 **動態設定** `promsrape.cluster.memberNum`，這樣每個 `vmagent` pod 啟動時都能獲得正確的 `memberNum`。



### 1.1. `entrypoint.sh` 內容

```
#!/bin/sh

# 設定 cluster memberNum, 使用環境變數, 如果沒設定則預設為 0
VMAGENT_MEMBER_NUM=${VMAGENT_MEMBER_NUM:-0}
VMAGENT_CLUSTER_SIZE=${VMAGENT_CLUSTER_SIZE:-3}

echo "Starting vmagent with cluster settings:"
echo "  Members Count: ${VMAGENT_CLUSTER_SIZE}"
echo "  Member Num: ${VMAGENT_MEMBER_NUM}"

exec /vmagent-prod \
  -promscrape.config=/vmagent/scrape.yml \
  -remoteWrite.url=http://vminsert:8480/insert/0/prometheus \
  -promscrape.cluster.membersCount=${VMAGENT_CLUSTER_SIZE} \
  -promscrape.cluster.memberNum=${VMAGENT_MEMBER_NUM} \
  "$@"
```

## 2. 當有 3 個 pod 時, `memberNum` 的分配

在 **Kubernetes StatefulSet** 中, 每個 `vmagent` pod 會有唯一的名稱, 例如:

```
vmagent-0
vmagent-1
vmagent-2
```

這些名稱可以用來動態設定 `VMAGENT\_MEMBER\_NUM`。

## 3. 如何讓 `entrypoint.sh` 自動設定 `memberNum`



在 `StatefulSet` 中，我們會設定：

```
env:
  - name: VMAGENT_MEMBER_NUM
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
```

這會讓 `VMAGENT\_MEMBER\_NUM` 取得 pod 的名稱，例如：

- `vmagent-0` 取得 `vmagent-0`
- `vmagent-1` 取得 `vmagent-1`
- `vmagent-2` 取得 `vmagent-2`

但這樣的變數值是 **字串**，我們需要 **取最後的數字** 來當作 `memberNum`。

## 4. 修改 `entrypoint.sh` 來解析 pod 名稱

```
#!/bin/sh

# 確保環境變數 VMAGENT_MEMBER_NUM 存在，否則預設為 vmagent-0
POD_NAME=${VMAGENT_MEMBER_NUM:-vmagent-0}

# 透過 shell 擷取 pod 名稱最後的數字 (memberNum)
VMAGENT_MEMBER_NUM=$(echo $POD_NAME | grep -oE '[0-9]+$')

# 設定 cluster 總數 (可用環境變數自訂，預設為 3)
VMAGENT_CLUSTER_SIZE=${VMAGENT_CLUSTER_SIZE:-3}

echo "Starting vmagent with cluster settings:"
echo "  Members Count: ${VMAGENT_CLUSTER_SIZE}"
echo "  Member Num: ${VMAGENT_MEMBER_NUM}"

exec /vmagent-prod \
```



```
-promscrape.config=/vmagent/scrape.yml \  
-remoteWrite.url=http://vminsert:8480/insert/0/prometheus \  
-promscrape.cluster.membersCount=${VMAGENT_CLUSTER_SIZE} \  
-promscrape.cluster.memberNum=${VMAGENT_MEMBER_NUM} \  
"$@"
```

## 5. pod 啟動時，`memberNum` 會是？

假設 `StatefulSet` 有 3 個 pod：

Pod 名稱	`memberNum` 設定
vmagent-0	0
vmagent-1	1
vmagent-2	2

這樣 `promscrape.cluster.memberNum` 會正確分配到 **0、1、2**，確保多個 `vmagent` 實例能夠協同運行。

## 6. Kubernetes `StatefulSet` 設定

在 `StatefulSet` 中，我們使用：

```
env:  
- name: VMAGENT_MEMBER_NUM  
  valueFrom:  
    fieldRef:  
      fieldPath: metadata.name
```



```
- name: VMAGENT_CLUSTER_SIZE  
  value: "3"
```

這樣 每個 pod 會自動根據名稱取得 `memberNum`，不用手動設定。

## 7. 如何驗證？

啟動 `vmagent` 後，執行：

```
kubectl logs -f vmagent-0 -n kube-vm  
kubectl logs -f vmagent-1 -n kube-vm  
kubectl logs -f vmagent-2 -n kube-vm
```

你應該會看到類似輸出：

```
Starting vmagent with cluster settings:  
Members Count: 3  
Member Num: 0
```

```
Starting vmagent with cluster settings:  
Members Count: 3  
Member Num: 1
```

```
Starting vmagent with cluster settings:  
Members Count: 3  
Member Num: 2
```

## 總結



- ✅ ``entrypoint.sh`` 會根據 **pod 名稱** 自動設定 ``memberNum``
- ✅ ``memberNum`` 分配為 **0, 1, 2**，確保分散工作負載
- ✅ ``StatefulSet`` 可自動管理 ``memberNum``，**無需手動修改配置**

這樣，**vmagent** 就能 **自動化擴展**，避免 ``promscrape.cluster.memberNum`` 需要手動配置的問題 🚀



參考"使用vmagent 代替Prometheus 採集監控指標- 技術顏良- 博客園.pdf"，以繁體中文詳細說明:# vmagent-

rbac.yaml

apiVersion: v1

kind: ServiceAccount

metadata:

name: vmagent

namespace: kube-vm

---

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: vmagent

rules:

- apiGroups: [ "", "networking.k8s.io", "extensions"]

resources:

- nodes

- nodes/metrics

- services

- endpoints

- endpointslices

- pods

- app

- ingresses

verbs: ["get", "list", "watch"]

- apiGroups: [ "" ]

resources:

- namespaces

- configmaps

verbs: ["get"]







```
- nonResourceURLs: ["/metrics", "/metrics/resources"]
  verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: vmagent
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: vmagent
subjects:
- kind: ServiceAccount
  name: vmagent
  namespace: kube-vm
```



## vmagent RBAC 權限設定 (vmagent-rbac.yaml) 詳細解析

在 Kubernetes 中，`vmagent` 需要從 **Kubernetes API Server** 獲取資源資訊（例如節點、Pod、服務等），因此需要適當的 **RBAC (Role-Based Access Control)** 設定，讓 `vmagent` 擁有相應的存取權限。

### 1. vmagent-rbac.yaml 主要包含的資源

這份 YAML 定義了：

1. **ServiceAccount** → 讓 `vmagent` 具備 Kubernetes API 存取身份
2. **ClusterRole** → 定義 `vmagent` 可以存取哪些 Kubernetes 資源
3. **ClusterRoleBinding** → 讓 `vmagent` 的 `ServiceAccount` 擁有 `ClusterRole` 權限

## 2. ServiceAccount 設定





```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: vmagent
  namespace: kube-vm
```

- **ServiceAccount**（服務帳戶）是 Kubernetes 中用來標識 Pod 執行時的身份。
- ``name: vmagent`` → 為 ``vmagent`` 指定一個服務帳戶。
- ``namespace: kube-vm`` → 這個 ServiceAccount 只屬於 ``kube-vm`` 命名空間（需確保 ``kube-vm`` 存在）。

## ServiceAccount 的作用

- 當 ``vmagent`` 在 Kubernetes **Pod** 內運行時，它會 **自動綁定這個 ServiceAccount**，然後 ``vmagent`` 便能透過 Kubernetes API 存取資源。

## 3. ClusterRole 設定

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: vmagent
rules:
  - apiGroups: [ "", "networking.k8s.io", "extensions" ]
    resources:
      - nodes
      - nodes/metrics
      - services
      - endpoints
      - endpointslices
      - pods
      - app
      - ingresses
```



```
verbs: ["get", "list", "watch"]
- apiGroups: [""]
resources:
  - namespaces
  - configmaps
verbs: ["get"]
- nonResourceURLs: ["/metrics", "/metrics/resources"]
verbs: ["get"]
```

## ClusterRole 的作用

- `ClusterRole` 允許 `vmagent` 存取 **Kubernetes API** 的各種資源。
- 這些權限適用於整個 **Kubernetes 叢集**，而不僅限於單一命名空間。

## ClusterRole 中的 `rules` 說明

資源類型	<code>apiGroups</code>	可存取的資源 ( <code>resources</code> )	操作 ( <code>verbs</code> )	作用
Kubernetes 主要資源	<code>""</code>	<code>nodes</code> , <code>nodes/metrics</code> , <code>services</code> , <code>endpoints</code> , <code>pods</code>	<code>get</code> , <code>list</code> , <code>watch</code>	允許 <code>vmagent</code> 獲取 節點、Pod、服務的 監控數據
網路相關資源	<code>networking.k8s.io</code> , <code>extensions</code>	<code>endpointslices</code> , <code>ingresses</code> , <code>app</code>	<code>get</code> , <code>list</code> , <code>watch</code>	獲取 Kubernetes 內 部服務的監控數據
其他系統資源	<code>""</code>	<code>namespaces</code> , <code>configmaps</code>	<code>get</code>	允許 <code>vmagent</code> 存取 命名空間與 ConfigMap
非資源 URL	<code>""</code>	<code>/metrics</code> , <code>/metrics/resources</code>	<code>get</code>	允許 <code>vmagent</code> 存取 <code>/metrics</code> 端點 (Prometheus 格式 指標)



## 為何需要這些權限？

- **監控節點 (nodes/metrics)**：允許 `vmagent` 從 Kubernetes 節點收集指標（如 CPU、記憶體使用率）。
- **監控 Pod (pods)**：監控特定 Pod 的資源消耗。
- **監控 Service (services, endpoints, endpointslices)**：監控 Kubernetes 服務的網路流量。
- **監控 Ingress (ingresses)**：監控 Kubernetes 入口流量。
- **存取 `/metrics`**：確保 `vmagent` 能夠收集 **Kubernetes 內建的監控數據**（如 API Server、kubelet）。

## 4. ClusterRoleBinding 設定

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

### ClusterRoleBinding 的作用

- 將 `vmagent` 的 **ServiceAccount** 綁定到 `vmagent` **ClusterRole**。
- 這樣 `vmagent` 服務帳戶就能存取 **Kubernetes API Server** 的監控數據。



## 5. 如何部署 `vmagent-rbac.yaml`

執行以下指令來套用 RBAC 設定：

```
kubectl apply -f vmagent-rbac.yaml
```

驗證 `ServiceAccount` 是否建立：

```
kubectl get serviceaccount -n kube-vm
```

應該會看到：

NAME	SECRETS	AGE
default	1	10d
vmagent	1	1m

驗證 `ClusterRole` 是否建立：

```
kubectl get clusterrole vmagent
```

驗證 `ClusterRoleBinding` 是否建立：

```
kubectl get clusterrolebinding vmagent
```

## 6. 總結

項目

用途

`ServiceAccount`

讓 `vmagent` Pod 有權限存取 Kubernetes API



項目	用途
<code>`ClusterRole`</code>	定義 <code>`vmagent`</code> 能存取哪些 Kubernetes 資源
<code>`ClusterRoleBinding`</code>	將 <code>`ServiceAccount`</code> 綁定到 <code>`ClusterRole`</code>

這樣 ``vmagent`` 就能安全地收集 **Kubernetes** 監控數據，並傳送到遠端 **VictoriaMetrics** 或 **Prometheus** 儲存系統！🚀



以繁體中文詳細說明:ServiceAccount、ClusterRole、ClusterRoleBinding



在 **Kubernetes** 中，RBAC（角色型存取控制，**Role-Based Access Control**）是一種授權機制，它允許管理者 **控制 Pod 和 ServiceAccount 的權限**，以確保系統安全性與最小權限原則（**Least Privilege Principle**）。

當我們使用 ``vmagent`` 來監控 Kubernetes 叢集時，它需要從 **API Server** 獲取節點、Pod、服務等監控數據，因此必須透過 **RBAC** 來賦予它必要的權限。

在這個過程中，我們主要使用三個 **RBAC 物件**：

1. **ServiceAccount**（服務帳戶）
2. **ClusterRole**（叢集角色）
3. **ClusterRoleBinding**（叢集角色綁定）

# 1. ServiceAccount（服務帳戶）

什麼是 **ServiceAccount**？





- **ServiceAccount** 是 **Pod** 在 **Kubernetes** 叢集內執行時的身份。
- 預設情況下，Kubernetes 會為每個 **命名空間（namespace）** 建立一個 ``default`` 的 ``ServiceAccount``，但如果 Pod 需要存取 API Server，則應 **自訂 ServiceAccount** 並綁定適當的權限。
- ``ServiceAccount`` 允許 Pod 透過 **JWT Token** 與 Kubernetes API 進行身份驗證。

## vmagent 的 ServiceAccount 設定

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: vmagent
  namespace: kube-vm
```

## 說明

欄位	說明
<code>`kind: ServiceAccount`</code>	創建一個 <code>`ServiceAccount`</code> 物件
<code>`metadata.name: vmagent`</code>	<code>`vmagent`</code> 這個服務帳戶的名稱
<code>`metadata.namespace: kube-vm`</code>	此帳戶限定在 <code>`kube-vm`</code> 命名空間

## 作用

- 讓 ``vmagent`` 使用 ``vmagent`` 這個 **ServiceAccount**（而不是預設的 ``default``）。
- 透過 **RBAC** 綁定適當的權限（例如讀取 Kubernetes 監控數據）。
- 避免 **Pod** 共享 ``default`` 帳戶，提高安全性。



## 2. ClusterRole (叢集角色)

### 什麼是 ClusterRole ？

- **ClusterRole** 是 **Kubernetes RBAC** 權限控制的角色，它定義 哪些 **Kubernetes** 資源可以被存取，以及 可以執行哪些操作 (**get**、**list**、**watch** 等)。
- **ClusterRole** 是全叢集範圍的角色，適用於多個命名空間。

### vmagent 的 ClusterRole 設定

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: vmagent
rules:
  - apiGroups: [ "", "networking.k8s.io", "extensions" ]
    resources:
      - nodes
      - nodes/metrics
      - services
      - endpoints
      - endpointslices
      - pods
      - app
      - ingresses
    verbs: [ "get", "list", "watch" ]
  - apiGroups: [ "" ]
    resources:
      - namespaces
      - configmaps
    verbs: [ "get" ]
  - nonResourceURLs: [ "/metrics", "/metrics/resources" ]
    verbs: [ "get" ]
```

### 說明





欄位	說明
<code>`kind: ClusterRole`</code>	建立一個叢集範圍的角色
<code>`metadata.name: vmagent`</code>	角色名稱為 <code>`vmagent`</code>
<code>`rules.apiGroups: ["" ]`</code>	代表管理 <b>核心 API</b> (如節點、 <b>Pod</b> 、 <b>Service</b> )
<code>`rules.apiGroups: ["networking.k8s.io", "extensions"]`</code>	允許存取 Kubernetes <b>網路相關資源</b> (如 <b>Ingress</b> )
<code>`rules.resources: [... ]`</code>	定義 <code>`vmagent`</code> 可以存取哪些 <b>Kubernetes 資源</b>
<code>`rules.verbs: ["get", "list", "watch"]`</code>	允許 <code>`vmagent`</code> <b>獲取、列出、監聽</b> 這些資源
<code>`nonResourceURLs: ["/metrics", "/metrics/resources"]`</code>	允許 <code>`vmagent`</code> <b>存取</b> <code>`/metrics`</code> <b>監控數據</b>

## 作用

- 允許 ``vmagent`` 存取 **Kubernetes 節點、Pod、服務等資源**，用來收集監控指標。
- 允許 ``vmagent`` 存取 ``/metrics`` 端點，用來抓取 Kubernetes 內部監控數據 (例如 Kubelet、API Server)。
- **支援自動發現 (Service Discovery)**，讓 ``vmagent`` 可以監控動態變化的 Pod 和 Service。

# 3. ClusterRoleBinding (叢集角色綁定)

## 什麼是 ClusterRoleBinding ?



- **ClusterRoleBinding** 是將 `ServiceAccount` 與 `ClusterRole` 綁定在一起，讓 `ServiceAccount` 繼承 `ClusterRole` 定義的權限。
- 這樣 `vmagent` 才能存取 Kubernetes API 獲取監控指標。

## vmagent 的 ClusterRoleBinding 設定

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

## 說明

欄位	說明
<code>kind: ClusterRoleBinding`</code>	建立一個 <code>ClusterRoleBinding`</code> 物件
<code>metadata.name: vmagent`</code>	綁定名稱為 <code>vmagent`</code>
<code>roleRef.apiGroup: rbac.authorization.k8s.io`</code>	指定使用 RBAC
<code>roleRef.kind: ClusterRole`</code>	綁定的是 <code>ClusterRole`</code>
<code>roleRef.name: vmagent`</code>	綁定 <code>vmagent` ClusterRole</code>
<code>subjects.kind: ServiceAccount`</code>	指定綁定 <code>ServiceAccount`</code>
<code>subjects.name: vmagent`</code>	綁定的 <code>ServiceAccount`</code> 名稱



欄位	說明
<code>`subjects.namespace: kube-vm`</code>	限定 <code>`vmagent`</code> ServiceAccount 在 <code>`kube-vm`</code> 命名空間內

## 作用

- 將 ``vmagent`` 的 ``ServiceAccount`` 授權給 ``vmagent`` **ClusterRole**。
- 確保 ``vmagent`` 能夠讀取 **Kubernetes** 監控數據，並發送到 **VictoriaMetrics** 或 **Prometheus**。
- **RBAC** 安全控制，避免 ``vmagent`` 存取不必要的資源，符合 **最小權限原則**。

# 4. 如何部署這些 RBAC 設定？

執行以下指令：

```
kubectl apply -f vmagent-rbac.yaml
```

檢查 ServiceAccount：

```
kubectl get serviceaccount -n kube-vm
```

檢查 ClusterRole：

```
kubectl get clusterrole vmagent
```

檢查 ClusterRoleBinding：



```
kubectl get clusterrolebinding vmagent
```

## 5. 總結

物件類型	用途
<b>ServiceAccount</b>	讓 `vmagent` 有 Kubernetes API 存取身份
<b>ClusterRole</b>	定義 `vmagent` 可以存取哪些 Kubernetes 資源
<b>ClusterRoleBinding</b>	將 `ServiceAccount` 綁定到 `ClusterRole`，使其生效

這樣，`vmagent` 就能安全地收集 **Kubernetes 監控數據**，並確保 **最小權限原則**，提高系統安全性與靈活性 🚀！



參考"使用vmagent 代替Prometheus 採集監控指標- 技術顏良- 博客園.pdf"，以繁體中文詳細說明:# vmagent-sts.yaml 的 args:

- -promscrape.config=/config/scrape.yml
- -remoteWrite.tmpDataPath=/tmpData，我該怎麼自己建立pvc、configmap，這2種資源各vmagent pod使用同一個還是不同個



在 `vmagent-sts.yaml` 中，`args` 參數設定了：

- ` -promscrape.config=/config/scrape.yml` → 使用 `ConfigMap` 掛載 `scrape.yml` 設定檔



- `-remoteWrite.tmpDataPath=/tmpData`` → 使用 `PersistentVolumeClaim (PVC)`` 持久化快取資料

## 1. `ConfigMap`` 與 `PersistentVolumeClaim (PVC)`` 的作用

資源類型	用途	StatefulSet 的使用方式
<b>ConfigMap</b>	存放 <code>vmagent`</code> 的監控配置 ( <code>scrape.yml`</code> )	所有 <code>vmagent` Pod</code> 共享同一個 <code>ConfigMap`</code>
<b>PersistentVolumeClaim (PVC)</b>	持久化 <code>vmagent`</code> 緩存的指標數據	每個 <code>vmagent` Pod</code> 需要獨立的 <code>PVC`</code>

### ◆ 為什麼 `ConfigMap`` 共享，但 `PVC`` 不能共享？

- `ConfigMap`` 存放的是靜態配置檔案，所有 `vmagent` Pod` 都需要相同的 `scrape.yml``，所以可以 共享。
- `PVC`` 存放的是每個 `vmagent`` 實例緩存的數據，不同 `Pod`` 會寫入不同的數據，不能共享，因此 每個 `Pod`` 需要自己的 `PVC``。

## 2. 建立 `ConfigMap`` (`vmagent-config.yaml``)

這個 `ConfigMap`` 會存放 `scrape.yml`` 配置檔，然後掛載到 `vmagent` Pod` 中。

`vmagent-config.yaml``

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: vmagent-config
  namespace: kube-vm
data:
```



```
scrape.yml: |
  global:
    scrape_interval: 15s
    scrape_timeout: 15s
  scrape_configs:
    - job_name: "kubernetes-nodes"
      kubernetes_sd_configs:
        - role: node
      relabel_configs:
        - source_labels: [__address__]
          regex: "(.?):10250"
          replacement: "${1}:9111"
          target_label: __address__
          action: replace
```

## 套用 `ConfigMap`

```
kubectl apply -f vmagent-config.yaml
```

## 如何掛載 `ConfigMap` ?

在 `StatefulSet` 的 `volumes` 和 `volumeMounts` 中掛載：

```
volumeMounts:
  - name: config
    mountPath: /config

volumes:
  - name: config
    configMap:
      name: vmagent-config
```

這樣 `vmagent` 就能夠從 `/config/scrape.yml` 讀取 `scrape.yml` 配置。



### 3. 建立 PersistentVolumeClaim (PVC)

每個 `vmagent` Pod 需要自己的 `PVC`，所以要在 `StatefulSet` 中使用 `volumeClaimTemplates`。

#### PVC 配置 (由 StatefulSet 自動產生)

在 `vmagent-sts.yaml` 中：

```
volumeClaimTemplates:
  - metadata:
      name: tmpdata
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
      storageClassName: nfs-client
```

#### PVC 的作用

- `tmpdata` 這個 PVC 會對應到 `-remoteWrite.tmpDataPath=/tmpData`。
- `ReadWriteOnce` (**RWO**) 表示每個 PVC 只能被 **一個 Pod 掛載**。
- `storageClassName: nfs-client` → 需要指定 Kubernetes 叢集內的儲存類型 (可視環境修改)。
- 每個 Pod 都會有自己獨立的 PVC，名稱格式類似：
  - `vmagent-0-tmpdata`
  - `vmagent-1-tmpdata`
  - `vmagent-2-tmpdata`



## 4. 在 StatefulSet (vmagent-sts.yaml) 掛載 ConfigMap 和 PVC

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: vmagent
  namespace: kube-vm
spec:
  replicas: 3
  serviceName: vmagent
  selector:
    matchLabels:
      app: vmagent
  template:
    metadata:
      labels:
        app: vmagent
    spec:
      serviceAccountName: vmagent
      containers:
        - name: agent
          image: victoriametrics/vmagent:v1.77.0
          args:
            - -promscrape.config=/config/scrape.yml
            - -remoteWrite.tmpDataPath=/tmpData
            - -remoteWrite.url=http://vminsert:8480/insert/0/prometheus
          ports:
            - name: http
              containerPort: 8429
          volumeMounts:
            - name: tmpdata
              mountPath: /tmpData
            - name: config
              mountPath: /config
      volumes:
        - name: config
          configMap:
            name: vmagent-config
```





```
volumeClaimTemplates:
  - metadata:
      name: tmpdata
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
      storageClassName: nfs-client
```

## 5. 總結

資源類型	用途	各 Pod 是否共用？	如何掛載？
<b>ConfigMap</b> (`vmagent-config`)	存放 `scrape.yml` 監控設定	✅ 所有 `vmagent` <b>Pod</b> 共享	掛載到 `/config/scrape.yml`
<b>PVC</b> (`tmpdata`)	持久化 `vmagent` 的快取數據	❌ 每個 `vmagent` <b>Pod</b> 需要獨立 <b>PVC</b>	掛載到 `/tmpData`

### 如何套用這些設定？

```
kubectl apply -f vmagent-config.yaml
kubectl apply -f vmagent-sts.yaml
```

### 如何確認 PVC 是否建立成功？

```
kubectl get pvc -n kube-vm
```

應該會看到：



NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS
AGE					
tmpdata-vmagent-0 5m	Bound	pvc-xxxxxxx	1Gi	RW0	nfs-client
tmpdata-vmagent-1 5m	Bound	pvc-yyyyyy	1Gi	RW0	nfs-client
tmpdata-vmagent-2 5m	Bound	pvc-zzzzzz	1Gi	RW0	nfs-client

## 6. 常見問題

### ? 為什麼 ConfigMap 可以共享，但 PVC 不能？

- **ConfigMap** 存放的是靜態的 ``scrape.yml`` 設定，所有 ``vmagent`` Pod 讀取的內容相同，因此可以 共享。
- **PVC** 存放的是 ``vmagent`` 產生的緩存數據，不同 ``vmagent`` Pod 可能會寫入不同數據，因此 必須獨立。

### ? 如果不想用 PVC，可以怎麼做？

如果不需要 ``remoteWrite.tmpDataPath`` 持久化數據，可以使用 ``emptyDir``：

```
volumes:  
  - name: tmpdata  
    emptyDir: {}
```

這樣，每次 ``vmagent`` Pod 重啟時，數據都會 重新初始化，不會持久化。

## 7. 結論



- **ConfigMap** → 存放 `scrape.yml`，所有 `vmagent` Pod 共用。
- **PVC** → 存放快取數據，每個 `vmagent` Pod 需要獨立的 PVC。
- **StatefulSet** `volumeClaimTemplates` 會自動為每個 `vmagent` Pod 建立獨立 PVC。
- 確保 PVC 名稱唯一，例如 `tmpdata-vmagent-0`，`tmpdata-vmagent-1`。

這樣，你的 `vmagent` 就能夠 正確管理監控指標數據，並確保高可用性與數據完整性 🚀！



一個pod可以使用幾個PVC



## 一個 Pod 可以使用幾個 PersistentVolumeClaim (PVC) ？

在 **Kubernetes** 中，一個 **Pod** 可以掛載 多個 **PersistentVolumeClaim (PVC)**，沒有硬性限制，實際可用數量取決於：

1. **Kubernetes 叢集設定**
2. **底層儲存系統 (StorageClass) 的支援**
3. **節點的資源限制 (如磁碟 IOPS、容量)**

通常，一個 Pod 可以掛載 多個 **PVC**，用來存儲不同類型的數據。例如：

- 一個 **PVC** 存儲應用程式日誌
- 一個 **PVC** 存儲資料庫數據
- 一個 **PVC** 存儲快取數據

---

## PVC 限制

1. `ReadWriteOnce (RW0)` 模式





- 一個 **PVC** 只能被一個 **Pod** 掛載。
- 適用於 **本地儲存或獨占式儲存**（如 **NFS**、**EBS**、**Ceph RBD**）。

## 2. `ReadWriteMany (RWX)` 模式

- 同一個 **PVC** 可以被多個 **Pod** 掛載。
- 適用於 **共享式儲存**（如 **NFS**、**CephFS**）。

## 3. `ReadOnlyMany (ROX)` 模式

- 多個 **Pod** 可以同時讀取，但不能寫入。

---

## 範例：一個 **Pod** 使用多個 **PVC**

以下是一個 **Pod** 同時掛載 **3 個 PVC** 的範例：

- `data-pvc`：儲存應用程式數據
- `logs-pvc`：儲存應用程式日誌
- `cache-pvc`：儲存快取數據

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
  namespace: default
spec:
  containers:
    - name: my-container
      image: my-app-image
      volumeMounts:
        - name: data-volume
          mountPath: /app/data
        - name: logs-volume
          mountPath: /app/logs
        - name: cache-volume
```



```
        mountPath: /app/cache
volumes:
  - name: data-volume
    persistentVolumeClaim:
      claimName: data-pvc
  - name: logs-volume
    persistentVolumeClaim:
      claimName: logs-pvc
  - name: cache-volume
    persistentVolumeClaim:
      claimName: cache-pvc
```

## 範例：StatefulSet 中的多個 PVC

在 `StatefulSet` 中，可以使用 `volumeClaimTemplates` 自動為每個 Pod 建立獨立的 PVC，例如：

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: my-stateful-app
  namespace: default
spec:
  replicas: 3
  serviceName: my-service
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-container
          image: my-app-image
```



```
    volumeMounts:
      - name: data-volume
        mountPath: /app/data
      - name: logs-volume
        mountPath: /app/logs
  volumeClaimTemplates:
    - metadata:
        name: data-volume
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 10Gi
        storageClassName: nfs-client
    - metadata:
        name: logs-volume
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 5Gi
        storageClassName: nfs-client
```

這樣，Kubernetes 會自動為 **每個 Pod 建立兩個 PVC**：

Pod 名稱	PVC 名稱
my-app-0	data-volume-my-app-0
my-app-0	logs-volume-my-app-0
my-app-1	data-volume-my-app-1
my-app-1	logs-volume-my-app-1
my-app-2	data-volume-my-app-2
my-app-2	logs-volume-my-app-2



## 總結

✅ 一個 Pod 可以使用多個 PVC，但具體數量取決於：

1. 底層存儲系統的支援
2. Kubernetes 叢集的設定
3. PVC 的 `AccessMode` (RWO、RWX、ROX)

✅ StatefulSet 可以使用 `volumeClaimTemplates` 自動建立多個 PVC。

✅ 如果 PVC 設定為 `ReadWriteOnce`，則每個 Pod 需要獨立的 PVC。如果使用 `ReadWriteMany`，則 PVC 可被多個 Pod 共享。



參考"使用vmagent 代替Prometheus 採集監控指標- 技術顏良- 博客園.pdf"，以繁體中文詳細說明:自動發現Kubernetes節點、# vmagent-config.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: vmagent-config
  namespace: kube-vm
data:
  scrape.yml: |
    global:
      scrape_interval: 15s
      scrape_timeout: 15s

    scrape_configs:
      - job_name: nodes
        kubernetes_sd_configs:
          - role: node
        relabel_configs:
          - source_labels: [__address__]
            regex: "(.?):10250"
            replacement: "${1}:9111"
```



```
target_label: __address__  
action: replace  
- action: labelmap  
  regex: __meta_kubernetes_node_label_(.+)
```



## vmagent 自動發現 Kubernetes 節點 (`kubernetes\_sd\_configs`)

在 Kubernetes 叢集中，**Pod**、**Service**、**Ingress**、**Node** 都是動態的，因此手動設定 `vmagent` 來監控這些資源並不現實。因此，我們可以透過 **Service Discovery**（自動發現機制）來讓 `vmagent` 自動發現 Kubernetes 內的監控目標，例如：

- 節點 (Node)
- Pod
- Service
- Ingress

在 `vmagent` 配置中，`kubernetes\_sd\_configs` (Kubernetes Service Discovery Configs) 用來自動發現這些資源。

### 1. `vmagent-config.yaml` 詳細解析

#### 完整 YAML

```
apiVersion: v1  
kind: ConfigMap  
metadata:  
  name: vmagent-config  
  namespace: kube-vm  
data:  
  scrape.yml: |
```





```
global:
  scrape_interval: 15s
  scrape_timeout: 15s

scrape_configs:
- job_name: nodes
  kubernetes_sd_configs:
    - role: node
  relabel_configs:
    - source_labels: [__address__]
      regex: "(.*):10250"
      replacement: "${1}:9111"
      target_label: __address__
      action: replace
    - action: labelmap
      regex: __meta_kubernetes_node_label_(.+)
```

## 2. `kubernetes\_sd\_configs` (自動發現 Kubernetes 資源)

這段設定：

```
kubernetes_sd_configs:
- role: node
```

- `kubernetes\_sd\_configs` 允許 `vmagent` 自動發現 Kubernetes 內的資源。
- `role: node` → 這個設定表示 `vmagent` 自動發現所有 Kubernetes 節點 (Node)，並收集它們的監控數據。

可用的 `role` 設定：



role 設定

值	作用
node	發現 Kubernetes 節點 (Node)，用來監控 每個節點的資源使用情況 (CPU、記憶體、網路等)
pod	發現 Kubernetes Pod，適合 監控應用程式的 Pod 指標
service	發現 Kubernetes Service，適合 監控 Service 相關的監控數據
endpoints	發現 Service Endpoints，適合 監控 Pod 與 Service 之間的流量
ingress	發現 Ingress，適合 監控 Ingress 進入流量的指標

在這個 vmagent-config.yaml 設定中，我們設定 role: node，因此 vmagent 會自動發現所有 Kubernetes 節點，並監控其指標。

### 3. relabel\_configs (重新標籤設定)

relabel\_configs 允許 vmagent 修改、刪除、替換指標的 Labels，以適應不同的監控需求。

#### (1) 調整 \_\_address\_\_ 標籤

```
- source_labels: [__address__]
  regex: "(.?):10250"
  replacement: "${1}:9111"
  target_label: __address__
  action: replace
```

- source\_labels: [\_\_address\_\_] → 讀取 \_\_address\_\_ (默認為 nodeIP:10250)
- regex: "(.?):10250" → 匹配 10250 端口 (Kubernetes kubelet 指標默認使用 10250 端口)



- ``replacement: "${1}:9111"`` → 將端口從 ``10250`` 替換為 ``9111``
- ``target_label: __address__`` → 修改 ``__address__`` 的值
- ``action: replace`` → 執行替換動作

為什麼要替換 ``10250`` → ``9111`` ？

- **Kubelet** 默認指標端口是 ``10250``，但這個端口可能無法直接被 ``vmagent`` 存取。
- 有些 **Kubernetes** 環境中，監控指標被代理到 ``9111`` 端口，所以這裡做了替換。

## (2) ``labelmap``（自動對應 Kubernetes 節點標籤）

```
- action: labelmap
  regex: __meta_kubernetes_node_label_(.+)
```

這個設定：

- 將 **Kubernetes** 節點的 ``labels`` 轉換成 ``Prometheus labels``，方便後續查詢和篩選。
- Kubernetes 節點標籤通常是 ``__meta_kubernetes_node_label_<label_key>``，例如：

```
__meta_kubernetes_node_label_environment=prod
__meta_kubernetes_node_label_region=us-east-1
```

- 這個 ``labelmap`` 設定會 自動去掉 ``__meta_kubernetes_node_label_`` 前綴，讓 ``vmagent`` 能直接使用這些標籤：

```
environment=prod
region=us-east-1
```

這樣的好處是：

- 在 **Grafana** 或 **Prometheus** 查詢時，可以根據 ``environment``、``region`` 來篩選節點監控數據。
- 例如：



```
node_cpu_seconds_total{environment="prod"}
```

## 4. 如何部署 `vmagent-config.yaml`

執行以下指令：

```
kubectl apply -f vmagent-config.yaml
```

查看 `ConfigMap` 是否建立成功：

```
kubectl get configmap -n kube-vm
```

應該會看到：

NAME	DATA	AGE
vmagent-config	1	5m

## 5. `StatefulSet` 如何掛載 `ConfigMap`

在 `StatefulSet` (`vmagent-sts.yaml`) 中，我們需要讓 `vmagent` 掛載 `ConfigMap` 作為 `scrape.yaml`：

```
containers:
- name: agent
  image: victoriametrics/vmagent:v1.77.0
  args:
  - -promscrape.config=/config/scrape.yaml
```



```
volumeMounts:
  - name: config
    mountPath: /config
volumes:
  - name: config
    configMap:
      name: vmagent-config
```

這樣 `vmagent` 會：

1. 從 `ConfigMap` 掛載 `/config/scrape.yml`
2. 讀取 `scrape.yml` 來自動發現 **Kubernetes** 節點

## 6. 如何驗證 `vmagent` 是否成功發現節點？

### 查詢 `vmagent` 日誌

執行：

```
kubectl logs -f vmagent-0 -n kube-vm
```

如果 `vmagent` **成功發現 Kubernetes 節點**，應該會看到：

```
{"ts":"2025-03-05T12:00:00.000Z","level":"info","caller":"VictoriaMetrics/lib/promscrape","msg":"Discovered new targets: ['10.1.1.10:9111', '10.1.1.11:9111', '10.1.1.12:9111']"}
```

### 查詢 `vmagent` 監控的目標

在瀏覽器打開：



```
http://vmagent-0.kube-vm.svc:8429/targets
```

應該能看到 `vmagent` 監控的 **Kubernetes** 節點列表。

## 7. 總結

設定	作用
<code>`kubernetes_sd_configs: role: node`</code>	讓 `vmagent` 自動發現 <b>Kubernetes</b> 節點 ( <b>Node</b> )
<code>`relabel_configs: __address__`</code>	將 `kubelet` 監控端口 `10250` 轉換為 `9111`
<code>`labelmap: __meta_kubernetes_node_label_(.+)`</code>	將節點標籤轉換為 <b>Prometheus Labels</b>
<code>`ConfigMap` 掛載到 `/config/scrape.yml`</code>	讓 `vmagent` 讀取配置
<code>`kubectl logs vmagent-0`</code>	驗證 `vmagent` 是否成功發現 <b>Kubernetes</b> 節點

透過這個 自動發現機制，`vmagent` 可以即時追蹤 **Kubernetes** 節點的變化，確保監控指標不會因為節點變動而遺失！🚀



以繁體中文詳細說明:3. relabel\_configs（重新標籤設定）



## `relabel\_configs`（重新標籤設定）詳細解析



``relabel_configs`` 是 **Prometheus** 和 **vmagent** 中的重要設定，它的主要用途是 **修改、刪除或新增監控指標的 Labels（標籤）**，以適應不同的監控需求。例如：

- 刪除不必要的標籤
- 修改標籤名稱或數值
- 根據條件篩選監控目標
- 統一標籤格式
- 修改 ``__address__`` 來指定監控端口

``relabel_configs`` 主要用於 **兩個地方**：

1. **抓取前（before scraping）**：在 ``scrape_configs`` 中應用，影響 ``vmagent`` 是否抓取該目標。
2. **發送前（before remote write）**：在 ``remote_write`` 中應用，影響 ``vmagent`` 發送到遠端存儲的數據。

## 1. ``relabel_configs`` 語法結構

```
relabel_configs:
- source_labels: [標籤1, 標籤2, ...]
  regex: "正則表達式"
  replacement: "替換後的值"
  target_label: "要修改的標籤"
  action: "動作類型"
```

### 可用的 ``action``（動作類型）

<code>`action`</code> 值	作用
<code>`replace`</code>	修改標籤值



<code>`action`</code> 值	作用
<code>`labelmap`</code>	批量修改標籤名稱
<code>`labeldrop`</code>	刪除指定標籤
<code>`labelkeep`</code>	僅保留符合條件的標籤
<code>`drop`</code>	刪除不符合條件的監控目標
<code>`keep`</code>	僅保留符合條件的監控目標

## 2. ``relabel_configs`` 例子解析

讓我們透過幾個 ``relabel_configs`` 的範例來理解它的使用方式。

### 2.1. 替換監控目標的 ``__address__``（端口變更）

設定

```
- source_labels: [__address__]
  regex: "(.*):10250"
  replacement: "${1}:9111"
  target_label: __address__
  action: replace
```

解析

設定項目	說明
<code>`source_labels: [__address__]`</code>	讀取監控目標的 <code>`__address__`</code> （格式類似 <code>`10.1.1.10:10250`</code> ）





設定項目	說明
<code>`regex: "(.*):10250"`</code>	匹配 <code>`__address__`</code> ，擷取 <code>`IP`</code> 部分，檢查是否是 <code>`10250`</code> 端口
<code>`replacement: "\${1}:9111"`</code>	將 <code>`10250`</code> 端口改為 <code>`9111`</code>
<code>`target_label: __address__`</code>	將結果寫回 <code>`__address__`</code>
<code>`action: replace`</code>	執行 <code>`replace`</code> 動作

## 應用場景

- **Kubernetes Kubelet** 預設的監控端口是 ``10250``，但有時候需要透過代理（如 ``node-exporter``）來改變端口，例如 ``9111``，這個 ``relabel_configs`` 就能完成這個轉換。

## 2.2. 自動映射 Kubernetes 節點標籤

### 設定

```
- action: labelmap
  regex: __meta_kubernetes_node_label_(.+)
```

### 解析

設定項目	說明
<code>`action: labelmap`</code>	批量修改標籤名稱
<code>`regex: __meta_kubernetes_node_label_(.+)`</code>	匹配所有 <code>`__meta_kubernetes_node_label_`</code> 開頭的標籤
結果	刪除 <code>`__meta_kubernetes_node_label_`</code> 前綴，轉換成普通標籤

範例 Kubernetes 內部節點的標籤原始格式：



```
__meta_kubernetes_node_label_environment=prod
__meta_kubernetes_node_label_region=us-east-1
```

這個 `relabel\_configs` 會轉換成標準的 **Prometheus Labels**：

```
environment=prod
region=us-east-1
```

## 應用場景

- 讓 `vmagent` 自動讀取 **Kubernetes** 節點的 **Labels**，方便在 **Grafana** 查詢時篩選節點。
- 例如在 Grafana 中：

```
node_cpu_seconds_total{environment="prod"}
```

## 2.3. 移除不必要的標籤

### 設定

```
- action: labeldrop
  regex: "instance"
```

### 解析

設定項目	說明
`action: labeldrop`	刪除符合條件的標籤
`regex: "instance"`	刪除 `instance` 標籤



## 應用場景

- 當 `instance` 標籤沒用時，可以刪除它，避免 Prometheus 儲存無用的標籤，提高效能。

## 2.4. 只保留特定標籤

### 設定

```
- action: labelkeep  
  regex: "region|environment"
```

### 解析

設定項目	說明
`action: labelkeep`	只保留符合條件的標籤
`regex: "region` environment"`	

## 應用場景

- 避免監控指標儲存太多無用的標籤，提高查詢效能。

## 2.5. 移除不需要的監控目標

### 設定

```
- source_labels: [__meta_kubernetes_pod_label_monitor]  
  regex: "false"  
  action: drop
```



## 解析

設定項目	說明
<code>`source_labels: [__meta_kubernetes_pod_label_monitor]`</code>	讀取 Pod 的 <code>`monitor`</code> 標籤
<code>`regex: "false"`</code>	如果 <code>`monitor=false`</code> ，則執行 <code>`drop`</code>
<code>`action: drop`</code>	刪除這個監控目標

## 應用場景

- 只監控 ``monitor=true`` 的 **Pod**，忽略不需要監控的 Pod。

## 3. ``relabel_configs`` 總結

作用	設定範例	應用場景
修改標籤值	<code>`replace`</code>	改變 <code>`__address__`</code> 端口
批量修改標籤名稱	<code>`labelmap`</code>	轉換 <b>Kubernetes</b> 節點標籤
刪除標籤	<code>`labeldrop`</code>	移除不必要的標籤（如 <code>`instance`</code> ）
保留特定標籤	<code>`labelkeep`</code>	只保留 <code>`region`</code> 、 <code>`environment`</code>
刪除監控目標	<code>`drop`</code>	忽略 <code>`monitor=false`</code> 的 <b>Pod</b>

## 4. 如何驗證 ``relabel_configs`` 是否生效？

可以透過 ``vmagent`` 提供的 ``targets`` 介面來檢查：



```
kubectl port-forward svc/vmagent 8429 -n kube-vm
```

然後在瀏覽器開啟：

```
http://localhost:8429/targets
```

這裡可以看到 `vmagent` 抓取了哪些目標，是否有被 `relabel\_configs` 影響。

## 5. 結論

- `relabel\_configs` 是 **vmagent** 監控設定的關鍵機制，它決定了哪些指標會被抓取、如何處理 Labels。
- 可以用來調整端口、刪除不必要的標籤、批量轉換 **Kubernetes Labels**，甚至忽略不必要的監控目標。
- 在 **Kubernetes** 監控中，適當的 `relabel\_configs` 設定能夠大幅減少儲存開銷，提升查詢效能！🚀



詳細說明:

```
volumeMounts:
  - name: tmpdata
    mountPath: /tmpData
  - name: config
    mountPath: /config與
args:
  - -promscrape.config=/config/scrape.yml
  - -remoteWrite.tmpDataPath=/tmpData 路徑的關係
```



`volumeMounts` 與 `args` 參數的路徑關係詳解



在 `vmagent` 的 `StatefulSet` 配置中，我們可以看到：

```
volumeMounts:
  - name: tmpdata
    mountPath: /tmpData
  - name: config
    mountPath: /config

args:
  - -promscrape.config=/config/scrape.yml
  - -remoteWrite.tmpDataPath=/tmpData
```

這些設定關係到 `vmagent` 如何讀取設定檔、以及如何使用 **PVC** 存儲快取數據，接下來我們詳細解析這些設定的作用與關聯。

## 1. `volumeMounts`：設定 `vmagent` 容器內的掛載點

`volumeMounts` 是 **Pod** 內部的目錄對應到 **Kubernetes Volume** 的掛載點，讓容器可以存取外部的 ConfigMap 或 PVC。

欄位	描述
`name: tmpdata`	對應到 <b>PVC</b> `tmpdata`，用來儲存 `vmagent` 的快取數據
`mountPath: /tmpData`	在容器內部掛載 `/tmpData`，讓 `vmagent` 可存取
`name: config`	對應到 <b>ConfigMap</b> `vmagent-config`，用來存放 `scrape.yml`
`mountPath: /config`	在容器內部掛載 `/config`，讓 `vmagent` 讀取 `scrape.yml` 設定檔



## 2. `args`：傳遞給 `vmagent` 的啟動參數

這些 `args` 參數控制 `vmagent` 啟動時的行為：

```
args:
  - -promscrape.config=/config/scrape.yml
  - -remoteWrite.tmpDataPath=/tmpData
```

參數	作用
<code>-promscrape.config=/config/scrape.yml`</code>	指定 `vmagent` 讀取 <code>/config/scrape.yml`</code> 作為監控配置
<code>-remoteWrite.tmpDataPath=/tmpData`</code>	指定 `vmagent` 使用 <code>/tmpData`</code> 來暫存未發送的監控數據

這兩個路徑對應到 `volumeMounts`，即：

- `/config/scrape.yml`` 來自 `ConfigMap`，包含 `vmagent` 的監控目標與 `scrape\_configs`。
- `/tmpData`` 來自 `PersistentVolumeClaim (PVC)`，用來存儲 遠端寫入 (remote write) 的暫存數據。

## 3. `volumeMounts` 與 `args` 的對應關係

`args` 參數	對應的 `volumeMounts`	來源
<code>/config/scrape.yml`</code>	<code>/config`</code>	`ConfigMap` (`vmagent-config`)
<code>/tmpData`</code>	<code>/tmpData`</code>	`PersistentVolumeClaim` (`tmpdata`)

對應 YAML



```
containers:
  - name: agent
    image: victoriametrics/vmagent:v1.77.0
    args:
      - -promscrape.config=/config/scrape.yml
      - -remoteWrite.tmpDataPath=/tmpData
    volumeMounts:
      - name: tmpdata
        mountPath: /tmpData
      - name: config
        mountPath: /config
volumes:
  - name: config
    configMap:
      name: vmagent-config
volumeClaimTemplates:
  - metadata:
      name: tmpdata
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
      storageClassName: nfs-client
```

## 4. `ConfigMap`（`/config/scrape.yml`）的作用

```
volumes:
  - name: config
    configMap:
      name: vmagent-config
```

- 這會將 `vmagent-config` 掛載到 `/config` 目錄，其中包含 `scrape.yml` 設定檔。





- `-promscrape.config=/config/scrape.yml`` 告訴 `vmagent`` 在 `/config/scrape.yml`` 讀取監控配置。

## 5. `PVC`` (`/tmpData``) 的作用

```
volumeClaimTemplates:
  - metadata:
      name: tmpdata
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
      storageClassName: nfs-client
```

- 這個 PVC 每個 `vmagent`` **Pod** 都有自己的 `tmpdata`` 卷。
- `-remoteWrite.tmpDataPath=/tmpData`` 讓 `vmagent`` 在 `/tmpData`` 目錄儲存遠端寫入的暫存數據。

## 6. 為什麼 `/tmpData`` 需要使用 PVC ?

- 避免數據遺失：當 `vmagent`` 無法即時發送數據到遠端存儲（如 `VictoriaMetrics``）時，這些數據會先存儲在 `/tmpData``，等連線恢復後再發送。
- 確保高可用性：如果 Pod 因為 重啟、遷移到不同節點 而變動，**PVC** 仍然能保持數據不遺失。
- 每個 **Pod** 都有自己的 `/tmpData``，確保數據獨立，避免不同 `vmagent`` **Pod** 之間互相影響。



## 7. 如何驗證 `volumeMounts` 是否成功掛載？

### (1) 檢查 ConfigMap 是否正確掛載

執行以下指令，確保 `scrape.yml` 存在：

```
kubectl exec -it vmagent-0 -- ls -l /config
```

應該會看到：

```
-rw-r--r-- 1 root root 354 Mar 6 10:00 scrape.yml
```

### (2) 檢查 PVC 是否正確掛載

執行以下指令，確保 `/tmpData` 目錄可用：

```
kubectl exec -it vmagent-0 -- ls -l /tmpData
```

如果 PVC 正確掛載，應該會看到類似：

```
total 0
```

(如果 `vmagent` 尚未寫入快取數據，目錄可能是空的)

### (3) 檢查 PVC 是否正確綁定

```
kubectl get pvc -n kube-vm
```

應該會看到：

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS
AGE					



tmpdata-vmagent-0 5m	Bound	pvc-xxxxxxx	1Gi	RW0	nfs-client
tmpdata-vmagent-1 5m	Bound	pvc-yyyyyy	1Gi	RW0	nfs-client
tmpdata-vmagent-2 5m	Bound	pvc-zzzzzz	1Gi	RW0	nfs-client

這代表 每個 `vmagent` **Pod** 都有自己的 **PVC**，確保數據獨立！

## 8. 總結

設定	作用	來源
<code>~/config/scrape.yml`</code>	`vmagent` 讀取的監控配置檔	`ConfigMap (vmagent-config)`
<code>~/tmpData`</code>	`vmagent` 儲存遠端寫入的暫存數據	`PersistentVolumeClaim (tmpdata)`
<code>promscrape.config=/config/scrape.yml`</code>	指定 `scrape.yml` 作為監控配置	`/config` 掛載的 `ConfigMap`
<code>-remoteWrite.tmpDataPath=/tmpData`</code>	設定暫存數據的存放目錄	`/tmpData` 掛載的 `PVC`

這樣的設計確保：

- `vmagent` 可以動態讀取 `scrape.yml` 配置
- 監控數據不會因為 **Pod** 重啟而丟失
- 所有 `vmagent` **Pod** 都有自己的快取目錄，避免數據混亂

這樣 `vmagent` 的監控架構更加穩定、靈活，適合 **大規模 Kubernetes 監控環境**！🚀