CI/CD

???????????

- Python Linter & Formatter
- GitHub Action Workflow
- Learning CI/CD
- Bitbucket Webhook

Python Linter & Formatter

Python ??????????

Linter

??? Linter

Linter ? lint??????????????——?????????????????

> Linter?????????????????????????????????? linter?? Python ???? linter ???pylint?pep8??? pycodestyle? ? flake8?

?? linter ?? Python ? package????? CLI ??????????????????

????????linter ????**????**?????????? **PEP 8** ???????????? linter ?????????????????????????????????????? **PEP 8 ??**???

Flake8 Linter

- GitHub: https://github.com/PyCQA/flake8
- Doc: https://flake8.pycqa.org/en/latest/
- VS Code ?? Python Linter?Formatter ?? - Code and Me

Formatter

??? Formatter

?????????????? linter???? linter ???????????????????? PEP 8 ???????????????? formatter????????????

?????formatter ??**????????????????**???????????**???**??

??????——??????????——?????? linter ?????????????????????Formatter ??????????????

??

Black Formatter

- GitHub: https://github.com/psf/black
- Doc: https://black.readthedocs.io/en/stable/index.html
- Python Flake8 ? Black Formatter ???????? - Code and Me
- ?? Black ??????——?Python ???????? - Code and Me

Ruff Formatter

- GitHub: https://github.com/astral-sh/ruff
- Doc: Ruff

GitHub Action Workflow

GitHub Actions ? GitHub ??? CI/CD ?????

???????

1. ??????2000 ??/?
2. ?????????

Tutorials

- GitHub Actions ?????? Lint?Format ? Type Check - Code and Me
- How to Build a Production-Ready DevOps Pipeline with Free Tools

Building Docker Image (workflow)

- Your Repo ? Settings ? Security ? Secrets and variables ? Actions
  - Repository secrets
    - Name: DOCKERHUB_TOKEN
    - Value: <*YOUR-TOKEN*>
  - Repository variables
    - Name: DOCKERHUB_USERNAME
    - Value: <*YOUR-USERNAME*>
- ?? Docker Registry ???????Docker Login · Actions · GitHub Marketplace

.github/workflows/deploy.yml :

```
name: Build and Push Docker Image


# ================================================================================
# 【觸發條件】
# ================================================================================
# - push: 當代碼推送到 master 分支時自動觸發
# - workflow_dispatch: 允許在 GitHub Actions 頁面手動觸發部署
# ================================================================================
on:
  #push:
  #  branches:
```

```
  #    - main
  workflow_dispatch:

# ================================================================================
# 【環境變數】
# ================================================================================
env:
  IMAGE_NAME: gemini-ocr-fastapi

jobs:
  build-and-push:
    runs-on: ubuntu-latest
    steps:
      # ------------------------------------------------------------------------
      # Step : 檢出代碼倉庫
      # ------------------------------------------------------------------------
      # 將 GitHub 倉庫的代碼下載到 runner 的工作目錄
      # 這是後續構建步驟的基礎
      # ------------------------------------------------------------------------
      - name: Checkout
        uses: actions/checkout@v4

      # ------------------------------------------------------------------------
      # Step : 釋放磁盤空間
      # ------------------------------------------------------------------------
      # GitHub Actions runner 的磁盤空間有限（約 14GB），為了確保構建過程順利進行，
      # 需要清理不必要的文件。此步驟會:
      # - 刪除 .NET SDK (如果不需要)
      # - 刪除 Android SDK (如果不需要)
      # - 刪除 GHC (Haskell 編譯器，如果不需要)
      # - 清理 Docker 系統 (鏡像、容器、卷等)
      # - 顯示磁盤使用情況
      #
      # 注意: docker system prune 有時可能導致不穩定，如果空間足夠可以註解掉
      # ------------------------------------------------------------------------
      - name: Free GitHub Actions Disk Space
        run: |
          sudo rm -rf /usr/share/dotnet
          sudo rm -rf /usr/local/lib/android
```

```
    sudo rm -rf /opt/ghc
    # 建議: prune 有時會導致不穩，如果空間還夠，可以先註解掉下面這行測試
    sudo docker system prune -af || true
    df -h

# ------------------------------------------------------------------------
# Step : 設置 Docker Buildx
# ------------------------------------------------------------------------
# Docker Buildx 是 Docker 的擴展構建工具，支持:
# - 多平台構建（如 linux/amd64, linux/arm64）
# - 構建緩存優化
# - 並行構建
#
# 配置說明:
# - image=moby/buildkit:latest: 使用最新版本的 buildkit 作為構建引擎
# - platforms: 聲明支持的平台（雖然這裡只構建 arm64，但保留 amd64 以備未來擴展）
# ------------------------------------------------------------------------
- name: Setup Docker Buildx
  uses: docker/setup-buildx-action@v2
  with:
    driver-opts: |
      image=moby/buildkit:latest
    platforms: linux/amd64,linux/arm64

# ------------------------------------------------------------------------
# Step : 登錄到 Docker Hub Registry
# ------------------------------------------------------------------------
# 在推送鏡像之前，必須先通過身份驗證登錄到 Docker Hub
#
# 認證方式:
# # - username/password: 從 GitHub Secrets 中讀取，確保敏感信息不會暴露在代碼中
#
# 安全提示: Docker Hub 憑證存儲在 GitHub Variables 與 Secrets 中，名稱為:
# - DOCKERHUB_USERNAME (Var)
# - DOCKERHUB_TOKEN  (Secret) 註: 使用 Personal Access Token
# ------------------------------------------------------------------------
- name: Log in to Docker Hub
  uses: docker/login-action@v3
  with:
```

```
          username: ${{ vars.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      # ------------------------------------------------------------------------
      # Step 5: 構建並推送 Docker 鏡像到 ACR
      # ------------------------------------------------------------------------
      # 這是核心構建步驟，負責：
      # 1. 使用 ./docker/Dockerfile 構建鏡像
      # 2. 將構建好的鏡像推送到 ACR
      #
      # 配置說明：
      # - context: 構建上下文目錄（整個倉庫根目錄）
      # - file: Dockerfile 的路徑
      # - push: true 表示構建完成後自動推送到 registry
      # - platforms: linux/arm64 表示構建 ARM64 架構的鏡像（適用於 Apple Silicon 或 ARM 服務器）
      # - tags: 鏡像標籤，使用 commit SHA 作為版本號，確保每次構建都有唯一標識
      #
      # 緩存策略：
      # - cache-from: 從 registry 拉取之前的構建緩存，加速構建過程
      # - cache-to: 將構建緩存推送到 registry，供下次構建使用
      # - mode=max: 使用最大緩存模式，保存所有構建層
      #
      # 鏡像標籤格式：stktrade.azurecr.io/stk-jixun-model:<commit-sha>
      # 例如：stktrade.azurecr.io/stk-jixun-model:abc123def456
      # ------------------------------------------------------------------------
      - name: Build and push Docker image
        uses: docker/build-push-action@v5
        with:
          context: ${{ github.workspace }}
          file: ./Dockerfile
          push: true
          platforms: linux/amd64
          tags: ${{ vars.DOCKERHUB_USERNAME }}/${{ env.IMAGE_NAME }}:${{ github.sha }}
          cache-from: type=registry,ref=${{ vars.DOCKERHUB_USERNAME }}/${{ env.IMAGE_NAME
}}:buildcache
          cache-to: type=registry,ref=${{ vars.DOCKERHUB_USERNAME }}/${{ env.IMAGE_NAME
}}:buildcache,mode=max

      # ------------------------------------------------------------------------
```

```
# Step 6: 生成鏡像構建摘要
# ------------------------------------------------------------------------
# 在 GitHub Actions 的 Summary 頁面生成一個美觀的 Markdown 報告，
# 包含:
# - 鏡像的完整信息（registry、名稱、標籤）
# - 如何手動拉取和運行鏡像的說明
# - 本次構建的元數據（commit SHA、workflow run ID）
#
# 這個摘要對於:
# - 快速查看構建結果
# - 手動測試特定版本的鏡像
# - 問題排查和版本追蹤
# 非常有用
# ------------------------------------------------------------------------
- name: Image Pull Summary
  run: |
    DOCKERHUB_NAME="${{ vars.DOCKERHUB_USERNAME }}"
    IMAGE_NAME="${{ env.IMAGE_NAME }}"
    IMAGE_TAG="${DOCKERHUB_NAME}/${IMAGE_NAME}:${{ github.sha }}"
    COMMIT_SHA="${{ github.sha }}"
    RUN_ID="${{ github.run_id }}"
    echo "## □□Image Build Summary" >> $GITHUB_STEP_SUMMARY
    echo "" >> $GITHUB_STEP_SUMMARY
    echo "### Image Information" >> $GITHUB_STEP_SUMMARY
    echo "- **Registry:** docker.io/\`${DOCKERHUB_NAME}\`" >> $GITHUB_STEP_SUMMARY
    echo "- **Image Name:** \`${IMAGE_NAME}\`" >> $GITHUB_STEP_SUMMARY
    echo "- **Tag:** \`${COMMIT_SHA}\`" >> $GITHUB_STEP_SUMMARY
    echo "- **Full Image:** \`${IMAGE_TAG}\`" >> $GITHUB_STEP_SUMMARY
    echo "" >> $GITHUB_STEP_SUMMARY
    echo "### How to Pull This Image" >> $GITHUB_STEP_SUMMARY
    echo "" >> $GITHUB_STEP_SUMMARY
    echo "1. **Login to Docker Hub:**" >> $GITHUB_STEP_SUMMARY
    echo "\`\`\`bash" >> $GITHUB_STEP_SUMMARY
    echo "docker login -u"  >> $GITHUB_STEP_SUMMARY
    echo "\`\`\`" >> $GITHUB_STEP_SUMMARY
    echo "" >> $GITHUB_STEP_SUMMARY
    echo "2. **Pull the image:**" >> $GITHUB_STEP_SUMMARY
    echo "\`\`\`bash" >> $GITHUB_STEP_SUMMARY
    echo "docker pull ${DOCKERHUB_NAME}/${IMAGE_TAG}" >> $GITHUB_STEP_SUMMARY
```

```
          echo "\`\`\`" >> $GITHUB_STEP_SUMMARY
          echo "" >> $GITHUB_STEP_SUMMARY
          echo "3. **Run the container:**" >> $GITHUB_STEP_SUMMARY
          echo "\`\`\`bash" >> $GITHUB_STEP_SUMMARY
          echo "docker run -d -e \"GOOGLE_API_KEY=你的金鑰\" -p 8000:8000 ${IMAGE_TAG}" >>
$GITHUB_STEP_SUMMARY
          echo "\`\`\`" >> $GITHUB_STEP_SUMMARY
          echo "" >> $GITHUB_STEP_SUMMARY
          echo "---" >> $GITHUB_STEP_SUMMARY
          echo "**Commit SHA:** \`${COMMIT_SHA}\` | **Workflow Run:** \`${RUN_ID}\`" >>
$GITHUB_STEP_SUMMARY
```

K8s deployment with Helm

- ???? (Bastion) ??? Helm ????
- ???GitHub Actions ? SSH to Bastion ? Execute Helm Upgrade ? K8s Cluster Updated

.github/workflows/deploy.yml :

```
jobs:
  # ============================================================================
  # Job 2: 使用 Helm 部署到 Kubernetes 集群
  # ============================================================================
  # 此 job 負責:
  # 1. 等待 build-and-deploy job 完成（確保鏡像已構建並推送）
  # 2. 通過 SSH 連接到 Azure Bastion 主機
  # 3. 在 Bastion 主機上執行 Helm 部署命令
  # 4. 更新 Kubernetes 集群中的應用程序
  #
  # 為什麼需要通過 Bastion?
  # - 安全考慮: Kubernetes 集群不直接暴露在公網
  # - 網絡隔離: 只有通過 Bastion 才能訪問集群
  # - 權限控制: Bastion 作為跳板機，集中管理訪問權限
  #
  # 部署流程:
  # GitHub Actions → SSH to Bastion → Execute Helm Upgrade → K8s Cluster Updated
  # ============================================================================
  helm-deploy:
    # 確保在 build-and-push job 成功完成後才執行
    needs: build-and-push
```

```
# 使用可重用的 Helm 部署 workflow
# 這個 workflow 定義在 ./.github/workflows/helm-deploy-reusable.yml
# 採用可重用 workflow 的好處：
# - 代碼復用：多個項目可以共享相同的部署邏輯
# - 易於維護：部署邏輯集中管理，修改一處即可影響所有使用它的 workflow
# - 標準化：確保所有項目的部署流程一致
uses: ./.github/workflows/helm-deploy-reusable.yml

# ========================================================================
# 【輸入參數】
# ========================================================================
# 這些參數會傳遞給可重用 workflow，用於配置部署行為
#
# - repo_path: 在 Bastion 主機上的倉庫路徑（用於拉取最新代碼或配置）
# - helm_release: Helm release 名稱，用於標識和管理部署的應用
# - helm_namespace: Kubernetes namespace，用於隔離不同環境的資源
#   （stktrade-prod 表示生產環境）
# - helm_chart_path: Helm chart 的本地路徑（在 Bastion 主機上）
# - helm_values_file: Helm values 文件路徑，包含部署配置（如資源限制、環境變數等）
# - kubeconfig_path: Kubernetes 配置文件路徑，用於認證和連接到集群
# - helm_set_values: 動態設置的 Helm values
#   image.tag=${{ github.sha }} 表示使用本次構建的鏡像版本（commit SHA）
#   這確保部署的是剛剛構建好的鏡像
# ========================================================================
with:
  repo_path: /home/alang/gemini-ocr-fastapi
  helm_release: gemini-ocr
  helm_namespace: my-devops-prod
  helm_chart_path: ~/gemini-ocr-fastapi/k8s/chart
  helm_values_file: ~/gemini-ocr-fastapi/k8s/chart/values.yaml
  kubeconfig_path: ~/my.kubeconfig
  helm_set_values: image.tag=${{ github.sha }}

# ========================================================================
# 【SSH 認證信息】
# ========================================================================
# 這些 secrets 存儲在 GitHub Secrets 中，用於 SSH 連接到 Bastion 主機
#
# - SSH_PRIVATE_KEY: SSH 私鑰，用於身份驗證
```

```
    # - SSH_HOST: Bastion 主機的 IP 地址或域名
    # - SSH_USER: SSH 登錄用戶名
    #
    # 安全提示：
    # - 所有敏感信息都存儲在 GitHub Secrets 中，不會暴露在代碼中
    # ========================================================================
    secrets:
      SSH_PRIVATE_KEY: ${{ secrets.MY_BASTION_KEY }}
      SSH_HOST: ${{ secrets.MY_BASTION_HOST }}
      SSH_USER: ${{ secrets.MY_BASTION_USERNAME }}
```

.github/workflows/helm-deploy-reusable.yml :

```
# ============================================================================
# GitHub Actions Reusable Workflow: Helm 部署可重用工作流
# ============================================================================
#
# 【什麼是可重用 Workflow？】
# 這是一個可重用的 workflow（reusable workflow），可以被其他 workflow 通過
# workflow_call 事件調用。類似於函數的概念，可以讓多個項目共享相同的部署邏輯。
#
# 【使用場景】
# 當多個項目需要執行相同的 Helm 部署流程時，可以：
# 1. 在各自的 workflow 中調用此可重用 workflow
# 2. 通過 inputs 參數傳入項目特定的配置
# 3. 避免重複編寫相同的部署代碼
#
# 【工作流程】
# 1. 通過 SSH 連接到遠程機器（通常是 Azure Bastion 或跳板機）
# 2. 更新遠程機器上的代碼倉庫到指定分支
# 3. 設置 Kubernetes 配置文件路徑
# 4. 構建並執行 Helm 部署命令
# 5. 清理環境（無論成功或失敗）
#
# 【優勢】
# - 代碼復用：多個項目共享同一套部署邏輯
# - 易於維護：修改一處即可影響所有使用它的 workflow
# - 標準化：確保所有項目的部署流程一致
# - 靈活性：通過參數化配置支持不同項目的需求
# ============================================================================
```

```yaml
name: Helm Deploy (Reusable)

# ================================================================================
# 【觸發方式】
# ================================================================================
# workflow_call: 表示此 workflow 可以被其他 workflow 調用
# 當其他 workflow 使用 uses: 關鍵字引用此 workflow 時，會觸發執行
# ================================================================================
on:
  workflow_call:
    # ============================================================================
    # 【輸入參數 (Inputs)】
    # ============================================================================
    # 這些參數由調用此 workflow 的父 workflow 傳入
    # 分為必需參數（required: true）和可選參數（required: false）
    # ============================================================================
    inputs:
      # ------------------------------------------------------------------------
      # 必需參數
      # ------------------------------------------------------------------------

      # repo_path: 遠程機器上的倉庫路徑
      # 用於定位需要更新的代碼倉庫位置
      # 例如: /home/azureuser/stk.jixun.model
      repo_path:
        description: 'Path to repository on remote machine'
        required: true
        type: string

      # helm_release: Helm release 名稱
      # 用於標識和管理 Kubernetes 中的應用部署
      # 同一個 namespace 中，release 名稱必須唯一
      helm_release:
        description: 'Helm release name'
        required: true
        type: string

      # helm_namespace: Kubernetes namespace
      # 用於隔離不同環境或項目的資源
```

```
      # 例如: stktrade-prod（生產環境）、stktrade-dev（開發環境）
      helm_namespace:
        description: 'Kubernetes namespace'
        required: true
        type: string

      # helm_chart_path: Helm chart 在遠程機器上的路徑
      # Chart 包含應用程序的部署模板和配置
      # 例如: ~/stk.jixun.model/k8s/chart
      helm_chart_path:
        description: 'Path to helm chart on remote machine'
        required: true
        type: string

      # helm_values_file: Helm values 文件路徑
      # Values 文件包含部署配置，如資源限制、環境變數、副本數等
      # 例如: ~/stk.jixun.model/k8s/chart/values.yaml
      helm_values_file:
        description: 'Path to values file on remote machine'
        required: true
        type: string

      # ------------------------------------------------------------------
      # 可選參數（SSH 連接相關）
      # ------------------------------------------------------------------

      # ssh_host: 目標機器的主機名或 IP 地址
      # 如果通過 secrets.SSH_HOST 提供，此參數可選
      # 優先級: secrets.SSH_HOST > inputs.ssh_host
      ssh_host:
        description: 'Target machine hostname or IP address (optional if SSH_HOST secret is
provided)'
        required: false
        type: string

      # ssh_user: SSH 登錄用戶名
      # 如果通過 secrets.SSH_USER 提供，此參數可選
      # 優先級: secrets.SSH_USER > inputs.ssh_user
      ssh_user:
        description: 'SSH username (optional if SSH_USER secret is provided)'
```

```yaml
  required: false
  type: string

# ssh_port: SSH 端口號
# 默認值為 22 (標準 SSH 端口)
# 如果目標機器使用非標準端口，可以通過此參數指定
ssh_port:
  description: 'SSH port'
  required: false
  type: string
  default: '22'

# ----------------------------------------------------------------------
# 可選參數 (Kubernetes 和 Helm 相關)
# ----------------------------------------------------------------------

# kubeconfig_path: Kubernetes 配置文件路徑
# Kubeconfig 文件包含集群連接信息和認證憑證
# 默認值: ~/.kube/config (Kubernetes 標準配置路徑)
kubeconfig_path:
  description: 'Kubeconfig path on remote machine'
  required: false
  type: string
  default: '~/.kube/config'

# helm_timeout: Helm 部署超時時間
# 如果部署在指定時間內未完成，Helm 會超時並失敗
# 默認值: 5m (5 分鐘)
# 格式: 數字 + 單位 (s=秒, m=分鐘, h=小時)
helm_timeout:
  description: 'Helm upgrade timeout'
  required: false
  type: string
  default: '5m'

# helm_wait: 是否等待部署完成
# true: 等待所有 Pod 就緒後才返回 (推薦用於生產環境)
# false: 提交部署後立即返回 (不等待 Pod 就緒)
# 默認值: true
helm_wait:
```

```
        description: 'Wait for deployment to complete'
        required: false
        type: boolean
        default: true

      # helm_set_values: 動態設置的 Helm values
      # 用於覆蓋 values.yaml 中的默認值
      # 格式: key1=value1,key2=value2 (逗號分隔)
      # 例如: image.tag=abc123,replicaCount=3
      #
      # 使用場景:
      # - 設置鏡像標籤 (如: image.tag=${{ github.sha }})
      # - 臨時調整副本數
      # - 覆蓋環境變數
      helm_set_values:
        description: 'Additional --set values (format: key1=value1,key2=value2)'
        required: false
        type: string

    # ================================================================================
    # 【Secrets (機密信息)】
    # ================================================================================
    # Secrets 用於存儲敏感信息，不會暴露在日誌中
    # 這些 secrets 由調用此 workflow 的父 workflow 傳入
    # ================================================================================
    secrets:
      # SSH_PRIVATE_KEY: SSH 私鑰 (必需)
      # 用於身份驗證，連接到遠程機器
      # 必須是與遠程機器上 authorized_keys 對應的私鑰
      SSH_PRIVATE_KEY:
        description: 'SSH private key for authentication'
        required: true

      # SSH_HOST: 目標機器主機名或 IP (可選)
      # 如果主機信息是敏感信息，可以通過 secret 傳入
      # 優先級高於 inputs.ssh_host
      SSH_HOST:
        description: 'Target machine hostname or IP address (optional, use if ssh_host input
is a secret)'
        required: false
```

```
      # SSH_USER: SSH 用戶名（可選）
      # 如果用戶名是敏感信息，可以通過 secret 傳入
      # 優先級高於 inputs.ssh_user
      SSH_USER:
        description: 'SSH username (optional, use if ssh_user input is a secret)'
        required: false

jobs:
  # ============================================================================
  # Job: Helm 部署
  # ============================================================================
  # 此 job 負責通過 SSH 連接到遠程機器並執行 Helm 部署
  # ============================================================================
  helm-deploy:
    runs-on: ubuntu-latest

    steps:
      # ----------------------------------------------------------------------
      # Step: 通過 SSH 執行 Helm 部署
      # ----------------------------------------------------------------------
      # 使用 appleboy/ssh-action 連接到遠程機器並執行部署腳本
      #
      # 執行流程:
      # 1. 建立 SSH 連接
      # 2. 在遠程機器上執行 script 中的命令
      # 3. 返回執行結果
      # ----------------------------------------------------------------------
      - name: Deploy with Helm
        uses: appleboy/ssh-action@v1
        with:
          # SSH 連接配置
          # 優先級: secrets > inputs
          # 這樣設計的好處是: 如果主機信息是敏感信息，可以通過 secret 傳入
          host: ${{ secrets.SSH_HOST || inputs.ssh_host }}
          username: ${{ secrets.SSH_USER || inputs.ssh_user }}
          key: ${{ secrets.SSH_PRIVATE_KEY }}
          port: ${{ inputs.ssh_port || '22' }}

          # 在遠程機器上執行的腳本
```

```
          # 注意：這些命令會在遠程機器上執行，而不是在 GitHub Actions runner 上
          script: |

            # ==================================================================
            # 【清理函數】
            # ==================================================================
            # 定義清理函數，用於在腳本退出時（無論成功或失敗）恢復倉庫狀態
            #
            # 為什麼需要清理？
            # - 部署過程中會切換到特定分支（如 master）
            # - 如果部署失敗，倉庫可能停留在錯誤的分支
            # - 清理函數確保倉庫恢復到默認分支（main 或 master）
            #
            # 清理邏輯：
            # 1. 檢查倉庫路徑是否存在
            # 2. 切換回默認分支（main 優先，如果不存在則嘗試 master）
            # 3. 使用 || true 確保即使切換失敗也不會中斷流程
            # ==================================================================
            cleanup() {
              if [ -d "${{ inputs.repo_path }}" ]; then
                cd "${{ inputs.repo_path }}" && git checkout main || git checkout master || true
              fi
            }

            # ==================================================================
            # 【註冊清理函數】
            # ==================================================================
            # 使用 trap 命令註冊清理函數，使其在腳本退出時自動執行
            # EXIT 信號會在腳本正常退出或異常退出時觸發
            #
            # 這類似於編程語言中的 finally 塊，確保清理邏輯一定會執行
            # ==================================================================
            trap cleanup EXIT

            # ==================================================================
            # 【步驟 1：更新代碼倉庫】
            # ==================================================================
            # 確保遠程機器上的代碼倉庫是最新的，並且切換到正確的分支
            #
```

```
# 執行流程：
# 1. cd 到倉庫目錄（如果失敗則退出）
# 2. git fetch origin: 從遠程倉庫獲取最新信息（不修改工作目錄）
# 3. git checkout -B: 創建或切換到指定分支，並追蹤遠程分支
#     - -B 表示如果分支存在則重置，不存在則創建
#     - "${{ github.ref_name }}" 是觸發 workflow 的分支名稱（如 master）
#     - "origin/${{ github.ref_name }}" 是遠程分支的引用
#
# 為什麼需要這一步？
# - 确保部署使用的是最新代码
# - 确保 Helm chart 和 values 文件是最新版本
# - 支持多分支部署（不同分支可能有不同的配置）
# ===================================================================
cd "${{ inputs.repo_path }}" || exit 1
git fetch origin
# 使用 reset --hard 强制更新所有文件，包括已修改的文件
git reset --hard "origin/${{ github.ref_name }}"

# ===================================================================
# 【步驟 2: 設置 Kubernetes 配置】
# ===================================================================
# 導出 KUBECONFIG 環境變數，告訴 kubectl 和 helm 使用哪個配置文件
#
# KUBECONFIG 的作用：
# - 指定 Kubernetes 集群的連接信息
# - 包含認證憑證和上下文信息
# - 允許訪問特定的 Kubernetes 集群
#
# 默認值：~/.kube/config（Kubernetes 標準配置路徑）
# 如果集群使用自定義配置文件，可以通過 kubeconfig_path 參數指定
# ===================================================================
export KUBECONFIG=${{ inputs.kubeconfig_path || '~/.kube/config' }}

# ===================================================================
# 【步驟 3: 構建基礎 Helm 命令】
# ===================================================================
# 構建 Helm upgrade --install 命令的基礎部分
#
# helm upgrade --install 說明：
# - --install: 如果 release 不存在則安裝，存在則升級
```

```
            #   這是一個智能命令，無需手動判斷是安裝還是升級
            #
            # 命令結構：
            # helm upgrade --install <release-name> <chart-path> \
            #   --namespace <namespace> \
            #   --values <values-file> \
            #   --timeout <timeout>
            #
            # 參數說明：
            # - ${{ inputs.helm_release }}: release 名稱（如：stk-jixun-model）
            # - ${{ inputs.helm_chart_path }}: chart 路徑（如：~/stk.jixun.model/k8s/chart）
            # - --namespace: 指定部署的 namespace
            # - --values: 指定 values 文件路徑
            # - --timeout: 設置超時時間（默認 5m）
            # ==================================================================
            HELM_CMD="helm upgrade --install ${{ inputs.helm_release }} ${{ inputs.helm_chart_path }} --namespace ${{ inputs.helm_namespace }} --values ${{ inputs.helm_values_file }} --timeout ${{ inputs.helm_timeout || '5m' }}"

            # ==================================================================
            # 【步驟 4：添加 --wait 選項（可選）】
            # ==================================================================
            # 如果 helm_wait 為 true，添加 --wait 標誌
            #
            # --wait 的作用：
            # - 等待所有 Pod 就緒後才返回
            # - 確保部署成功完成，而不僅僅是提交了部署請求
            # - 如果 Pod 無法就緒，會超時並失敗
            #
            # 為什麼需要條件判斷？
            # - helm_wait 是 boolean 類型，但在 shell 中會被轉換為字符串 "true" 或 "false"
            # - 需要字符串比較來判斷是否啟用
            #
            # 使用場景：
            # - 生產環境：通常設置為 true，確保部署成功
            # - 開發環境：可以設置為 false，快速返回
            # ==================================================================
            if [ "${{ inputs.helm_wait }}" = "true" ]; then
              HELM_CMD="${HELM_CMD} --wait"
            fi
```

```
# ============================================================================
# 【步驟 5: 添加 --set 值（可選）】
# ============================================================================
# 如果提供了 helm_set_values，解析並添加到命令中
#
# 處理邏輯:
# 1. 檢查 helm_set_values 是否為空
# 2. 如果非空，按逗號分割成數組
# 3. 遍歷數組，為每個值添加 --set 標誌
#
# 示例:
# 輸入: image.tag=abc123,replicaCount=3
# 輸出: --set image.tag=abc123 --set replicaCount=3
#
# IFS（Internal Field Separator）說明:
# - IFS=',' 設置字段分隔符為逗號
# - read -ra SET_VALUES 將字符串讀入數組
# - <<< 是 here-string，將變數內容作為輸入
#
# 使用場景:
# - 動態設置鏡像標籤（如: image.tag=${{ github.sha }}）
# - 臨時調整配置（如: replicaCount、資源限制等）
# - 覆蓋環境變數
# ============================================================================
HELM_SET_VALUES="${{ inputs.helm_set_values }}"
if [ -n "${HELM_SET_VALUES}" ]; then
  # 按逗號分割字符串為數組
  IFS=',' read -ra SET_VALUES <<< "${HELM_SET_VALUES}"
  # 遍歷數組，為每個值添加 --set 標誌
  for set_val in "${SET_VALUES[@]}"; do
    HELM_CMD="${HELM_CMD} --set ${set_val}"
  done
fi

# ============================================================================
# 【步驟 6: 執行 Helm 命令】
# ============================================================================
# 使用 eval 執行構建好的 Helm 命令
#
```

```
              # 為什麼使用 eval?
              # - HELM_CMD 是一個包含完整命令的字符串
              # - 需要將字符串解析為命令並執行
              # - eval 會先展開變數，然後執行命令
              #
              # 執行結果:
              # - 成功: Helm 會部署或升級應用，返回成功狀態碼
              # - 失敗: Helm 會返回錯誤信息和非零狀態碼，導致 workflow 失敗
              #
              # 注意:
              # - 如果使用了 --wait，會等待所有 Pod 就緒
              # - 如果超時，會返回超時錯誤
              # - 無論成功或失敗，trap 註冊的清理函數都會執行
              # ====================================================================
              eval $HELM_CMD
```

Manually run

```
# Create a secret for dockerhub-pull-secret
kubectl create secret docker-registry dockerhub-pull-secret \
  --docker-server=docker.io \
  --docker-username=alangtw \
  --docker-password='XXXXXXXXXXXXXXXXXXXX' \
  -n my-devops-prod

# Create a secret for gemini-ocr-secret
kubectl create secret generic gemini-ocr-api-secret \
  --from-literal=API_KEY='ThisIsTheAPKey' \
  --from-literal=GEMINI_API_KEY='XXXXXXXXXXXXXX' \
  -n my-devops-prod

# Deploy with helm
# Usage:
#  helm upgrade --install your-app-name k8s/chart \
#    --namespace your-namespace-prod \
#    --create-namespace
cd /path/to/your/repo
helm upgrade --install gemini-ocr k8s/chart --namespace my-devops-prod --values
k8s/chart/values.yml --timeout 5m
```

Build .deb package (workflow)

- workflow: https://github.com/zquestz/plank-reloaded/blob/master/.github/workflows/debian-release.yml
- Build and Release on Debian Bookworm
- Use Package-checking tool - Lintian

???? README ???

- workflow: https://github.com/hesamsheikh/awesome-openclaw-usecases/blob/main/.github/workflows/update-badge.yml

Learning CI/CD

Introduction

- Introduction to GitHub Actions for Python Projects - PyImageSearch

Drone CI

- Drone CI is a self-service Continuous Integration platform for busy development teams.
  - iT+: ???????? Container ?????? CI/CD ???

Jenkins

- Jenkins - A common open source CI system
  - [Day 15] Jenkins ?? - iT ???::???????????? IT ????

Codefresh

- Codefresh - The Best CI/CD Platform With GitOps - OSTechNix

Bitbucket Webhook

URL: How to Create a Basic CI/CD Pipeline with Webhooks on Linux

Tech Stacks

- Bitbucket
- Webhook
- Flask-based Python server