

單元 19：後端架構——Flask 與 Gemini API 整合

這是本書的最後一個單元。前一個單元介紹了前端架構，這個單元將介紹後端架構——Flask 框架和 Gemini API 的整合。

理解後端架構可以幫助你：

- 排除 API 相關的問題
- 客製化 AI 處理的邏輯
- 擴展系統支援其他 AI 服務商

什麼是後端？

在 Web 應用程式中，「後端」是在伺服器上執行的程式。它負責：

- **接收請求**：從前端接收 HTTP 請求
- **處理資料**：執行商業邏輯、呼叫外部 API
- **回傳結果**：把處理結果以 JSON 等格式回傳給前端

簡報編修系統的後端有兩個主要職責：

1. 呼叫 Gemini AI 進行文字移除和背景修復
2. 呼叫 Gemini AI 進行 OCR 文字提取

Flask 框架簡介

Flask 是一個輕量級的 Python Web 框架。它的設計理念是「微框架」——提供核心功能，但保持彈性，讓開發者可以選擇需要的擴充功能。

為什麼選擇 Flask？

輕量靈活：Flask 的核心很小，沒有強制的專案結構或依賴。你可以按照自己的方式組織程式碼。

易於學習：Flask 的 API 設計直覺，幾行程式碼就能建立一個可運作的 Web 應用。

豐富的擴充：Flask 有大量的擴充套件，可以按需加入功能，如資料庫整合、使用者認證等。

適合 API 開發：Flask 特別適合開發 RESTful API，這正是簡報編修系統後端的需求。

基本概念

一個最簡單的 Flask 應用：

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run()
```

這幾行程式碼就建立了一個可運作的 Web 伺服器。當你訪問

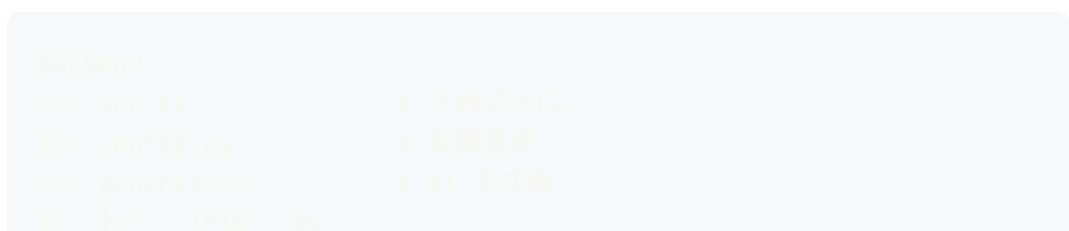
<http://localhost:5000/> 時，會看到「Hello, World!」。

關鍵概念：

- `Flask(__name__)`：建立 Flask 應用實例
- `@app.route('/')`：裝飾器，定義 URL 路徑與函式的對應
- `app.run()`：啟動開發伺服器

專案結構

讓我們看看簡報編修系統的後端結構：





app.py (主程式入口)

這是後端的入口點，負責建立和配置 Flask 應用：

```
from flask import Flask
from flask_cors import CORS

from .config import Config
from .routes import create_api_blueprint

def create_app() -> Flask:
    """創建 Flask 應用"""
    app = Flask(__name__)

    # 配置
    app.config['DEBUG'] = Config.DEBUG

    # CORS (跨域資源共享)
    CORS(app, origins=Config.CORS_ORIGINS)

    # 註冊 API 藍圖
    api_bp = create_api_blueprint()
    app.register_blueprint(api_bp)

    # 健康檢查
    @app.route('/health')
    def health():
        return {'status': 'ok', 'service': '亮言 NotebookLM 簡報編'

    return app

# 創建應用實例
app = create_app()

if __name__ == '__main__':
    app.run()
```

```
        host=Config.HOST,  
        port=Config.PORT,  
        debug=Config.DEBUG  
    )
```

關鍵點：

CORS：CORS (Cross-Origin Resource Sharing) 是瀏覽器的安全機制。因為前端 (port 5173) 和後端 (port 8099) 在不同的 port，瀏覽器會阻擋跨域請求。`CORS(app)` 允許前端跨域存取後端 API。

Blueprint：Blueprint 是 Flask 用來組織路由的方式。大型應用可以把不同功能的路由分開，然後用 `register_blueprint()` 註冊。

create_app() 模式：這是 Flask 的最佳實踐，稱為「應用工廠模式」。它讓應用的建立變得可配置，方便測試和部署。

config.py (配置管理)

這個模組負責管理系統的配置，包括讀取 `providers.yaml` 檔案：

```
import yaml  
  
class Config:  
    DEBUG = True  
    HOST = '0.0.0.0'  
    PORT = 8099  
    CORS_ORIGINS = ['http://localhost:5173']  
  
    @classmethod  
    def get_provider_config(cls):  
        """讀取 AI 服務商配置"""  
        with open('providers.yaml', 'r') as f:  
            config = yaml.safe_load(f)  
  
            active = config.get('active_provider', 'gemini')  
            return config.get('providers', {}).get(active, {})
```

配置的集中管理讓修改變得更容易，也避免了「魔術數字」散布在程式碼各處。

路由設計

Flask 使用「路由」(Route) 來定義 URL 與處理函式的對應。

處理路由 (process_routes.py)

這是系統最核心的 API，負責處理圖片：

```
from flask import Blueprint, request, jsonify, Response
import json

process_bp = Blueprint('process', __name__)

@process_bp.route('/process', methods=['POST'])
def process_pages():
    """處理選定的頁面"""
    data = request.get_json()
    pages = data.get('pages', [])

    def generate():
        """SSE 事件生成器"""
        for i, page in enumerate(pages):
            # 處理每一頁...
            yield f"event: progress\ndata: {json.dumps({...})}\n\n"

            yield f"event: finish\ndata: {json.dumps({...})}\n\n"

    return Response(
        generate(),
        mimetype='text/event-stream'
    )
```

關鍵點：

Blueprint：`Blueprint('process', __name__)` 建立一個藍圖，用於組織相關的路由。

POST 方法：`methods=['POST']` 指定這個路由只接受 POST 請求。POST 通常用於傳送資料給伺服器處理。

`request.get_json()`：從請求體中解析 JSON 資料。前端送來的頁面資訊就是透過這個方式接收。

SSE (Server-Sent Events)：這是一種讓伺服器主動推送資料給客戶端的技術。因為處理多頁可能需要較長時間，使用 SSE 可以即時回報進度，而不是等全部完成才回傳。

OCR 路由

```
@process_bp.route('/ocr', methods=['POST'])
def ocr_extract():
    """OCR 文字提取 API"""
    data = request.get_json()
    image_base64 = data.get('image', '')

    # 取得服務商配置
    provider_config = Config.get_provider_config()
    provider_type = provider_config.get('type', 'gemini')

    # 創建生成器
    generator = GeneratorFactory.create(provider_type, provider_c

    # 提取文字
    blocks = generator.ocr_extract_text(image_base64)

    return jsonify({
        "success": True,
        "blocks": blocks
    })
```

jsonify()：把 Python 字典轉換成 JSON 格式的 HTTP 回應。

AI 生成器架構

系統使用「策略模式」和「工廠模式」來設計 AI 生成器，讓它可以輕鬆支援不同的 AI 服務商。

抽象基類 (base.py)

```
from abc import ABC, abstractmethod
from typing import Dict, Any, List

class GeneratorBase(ABC):
    """AI 生成器抽象基類"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.api_key = config.get('api_key', '')

    @abstractmethod
```

```

def remove_text_from_image(self, image_base64: str) -> str:
    """移除圖片中的文字並修復背景"""
    pass

@abstractmethod
def ocr_extract_text(self, image_base64: str) -> List[Dict[str, Any]]:
    """從圖片中提取文字及其位置資訊"""
    pass

@abstractmethod
def validate_config(self) -> bool:
    """驗證配置是否有效"""
    pass

@abstractmethod
def test_connection(self) -> Dict[str, Any]:
    """測試 API 連線"""
    pass

```

ABC (Abstract Base Class)：Python 的抽象基類。帶有 `@abstractmethod` 裝飾器的方法必須被子類別實現。

這個設計的好處是：

- 定義了所有 AI 生成器必須實現的介面
- 保證所有實現都有一致的 API
- 方便新增其他 AI 服務商的支援

Gemini 生成器 (gemini.py)

這是 Gemini AI 的具體實現：

```

import google.generativeai as genai
from .base import GeneratorBase

class GeminiGenerator(GeneratorBase):
    """Gemini AI 生成器"""

    def __init__(self, config: Dict[str, Any]):
        super().__init__(config)
        genai.configure(api_key=self.api_key)
        self.model_image_edit = config.get('model_image_edit', 'g')
        self.model_text_gen = config.get('model_text_gen', 'gemin'

```

```

def remove_text_from_image(self, image_base64: str) -> str:
    """使用 Gemini 移除圖片中的文字"""
    prompt = "Edit this image: Erase ALL text completely. Inp

    # 呼叫 Gemini API...
    response = client.models.generate_content(
        model=self.model_image_edit,
        contents=[prompt, image_data],
        config=types.GenerateContentConfig(
            response_modalities=["IMAGE"]
        )
    )

    # 提取結果圖片
    return base64.b64encode(response.candidates[0].content.pa

def ocr_extract_text(self, image_base64: str) -> List[Dict[str, Any]]:
    """使用 Gemini 提取文字"""
    prompt = """Analyze this image and extract all text block
Return as JSON array with: text, box_2d, font_size_pt, fo

    model = genai.GenerativeModel(self.model_text_gen)
    response = model.generate_content([prompt, image_data])

    return json.loads(response.text)

```

工廠模式 (factory.py)

工廠模式用於建立生成器實例：

```

from .gemini import GeminiGenerator

class GeneratorFactory:
    """AI 生成器工廠"""

    @staticmethod
    def create(provider_type: str, config: dict):
        """根據類型創建生成器"""
        if provider_type == 'gemini':
            return GeminiGenerator(config)
        # 未來可以加入其他服務商
        # elif provider_type == 'openai':
        #     return OpenAIGenerator(config)

```

```
        else:  
            raise ValueError(f"不支援的服務商: {provider_type}")
```

這個設計讓新增其他 AI 服務商變得簡單：

1. 建立一個新的生成器類別，繼承 `GeneratorBase`
2. 實現所有抽象方法
3. 在工廠中加入對應的建立邏輯

Gemini API 整合

API 金鑰配置

Gemini API 需要 API Key 來認證。系統從 `providers.yaml` 讀取配置：

```
active_provider: gemini  
  
providers:  
    gemini:  
        type: gemini  
        api_key: "YOUR_API_KEY_HERE"  
        model_image_edit: gemini-2.5-flash-preview-05-20  
        model_text_gen: gemini-2.5-flash
```

文字移除 API

Gemini 的圖片編輯 API 可以根據文字提示修改圖片：

```
from google import genai as genai_client  
from google.genai import types  
  
def remove_text_from_image(self, image_base64: str) -> str:  
    prompt = "Edit this image: Erase ALL text completely. Inpaint  
  
    client = genai_client.Client(api_key=self.api_key)  
  
    response = client.models.generate_content(  
        model=self.model_image_edit,  
        contents=[  
            types.Content(
```

```

        role="user",
        parts=[
            types.Part.from_text(text=prompt),
            types.Part.from_bytes(data=image_data, mime_t
        ]
    )
],
config=types.GenerateContentConfig(
    response_modalities=["IMAGE"]
)
)

return base64.b64encode(response.candidates[0].content.parts[

```

關鍵點：

Prompt 設計：「Erase ALL text completely. Inpaint and restore the background seamlessly.」

這個提示詞告訴 AI：

1. 移除所有文字
2. 用 Inpainting 技術修復背景
3. 只輸出編輯後的圖片

****response_modalities=["IMAGE"]****：指定回應格式為圖片，而不是文字。

OCR API

Gemini 也可以用於 OCR（光學字元辨識）：

```

def ocr_extract_text(self, image_base64: str) -> List[Dict[str, A
prompt = """Analyze this image and extract all text blocks wi

For each text block, provide:
- text: The exact text content
- box_2d: [ymin, xmin, ymax, xmax] in 0-1000 coordinate syste
- font_size_pt: Estimated font size in points
- font_weight: "normal" or "bold"
- font_style: "normal" or "italic"
- text_align: "left", "center", or "right"
- color: Hex color code

Return as JSON array."""

```

```
model = genai.GenerativeModel(self.model_text_gen)
response = model.generate_content(
    [prompt, {"mime_type": "image/png", "data": image_data}],
    generation_config={"response_mime_type": "application/json"}
)

return json.loads(response.text)
```

關鍵點：

`**response_mime_type="application/json"**`：指定回應格式為 JSON，讓 AI 直接輸出結構化的資料。

座標系統：使用 0-1000 的座標系統，而不是像素。這樣不管圖片的實際尺寸，座標都是相對的，方便後續處理。

重試機制

API 呼叫可能會因為各種原因失敗，系統實現了重試機制：

```
def _call_with_retry(self, func, max_retries: int = 5):
    """帶重試機制的 API 調用"""
    delays = [2, 4, 8, 16, 32] # 指數退避延遲

    for i in range(max_retries):
        try:
            return func()
        except Exception as e:
            error_str = str(e).lower()

            # 429 速率限制 - 等待後重試
            if '429' in error_str or 'rate' in error_str:
                if i < max_retries - 1:
                    delay = delays[min(i, len(delays) - 1)]
                    time.sleep(delay)
                    continue

            # 不可重試的錯誤
            raise

    raise Exception(f"API 調用失敗，已重試 {max_retries} 次")
```

指數退避：每次重試的等待時間加倍（2, 4, 8, 16, 32 秒）。這是處理速率限制的最佳實踐，避免短時間內大量請求。

可重試 vs. 不可重試：

- 429（速率限制）、500（伺服器錯誤）、503（服務不可用）：這些是暫時性錯誤，可以重試
- 401（認證失敗）、400（請求錯誤）：這些是永久性錯誤，重試也不會成功

SSE (Server-Sent Events)

處理多頁時，系統使用 SSE 來即時回報進度。

什麼是 SSE？

SSE 是一種讓伺服器主動推送資料給客戶端的技術。與傳統的請求-回應模式不同，SSE 建立一個持續的連線，伺服器可以隨時發送事件。

Flask 中的 SSE

```
def process_pages():
    def generate():
        for i, page in enumerate(pages):
            # 發送進度事件
            yield f"event: progress\ndata: {json.dumps({'page': i

            # 處理頁面...

            # 發送完成事件
            yield f"event: complete\ndata: {json.dumps({'page': i

            # 發送結束事件
            yield f"event: finish\ndata: {json.dumps({'success': True

    return Response(
        generate(),
        mimetype='text/event-stream',
        headers={
            'Cache-Control': 'no-cache',
            'Connection': 'keep-alive'
```

```
    }  
)
```

關鍵點：

yield：使用 Python 的生成器（generator）來產生事件流。每次 **yield** 會發送一個事件給客戶端。

事件格式：SSE 的事件格式是：

```
data: {  
    type: 'progress'  
    value: 50  
}  
  
data: {  
    type: 'complete'  
    result: 'Success'  
}  
  
data: {  
    type: 'finish'  
}
```

注意最後要有兩個換行符號。

****mimetype='text/event-stream'****：告訴瀏覽器這是一個 SSE 流，而不是普通的 HTTP 回應。

前端接收 SSE

```
const eventSource = new EventSource('/api/process')  
  
eventSource.addEventListener('progress', (e) => {  
    const data = JSON.parse(e.data)  
    updateProgress(data.page, data.percent)  
})  
  
eventSource.addEventListener('complete', (e) => {  
    const data = JSON.parse(e.data)  
    addResult(data.page, data.result)  
})  
  
eventSource.addEventListener('finish', (e) => {  
    eventSource.close()  
    showResults()  
})
```

錯誤處理

良好的錯誤處理對於穩定的系統至關重要。

API 層面

```
@process_bp.route('/ocr', methods=['POST'])
def ocr_extract():
    try:
        # 驗證輸入
        data = request.get_json()
        if not data.get('image'):
            return jsonify({"success": False, "error": "未提供圖片"})

        # 處理...
        return jsonify({"success": True, "blocks": blocks})

    except Exception as e:
        logger.error(f"OCR 失敗: {e}")
        return jsonify({"success": False, "error": str(e)}), 500
```

關鍵點：

輸入驗證：在處理之前先檢查輸入是否有效。如果無效，回傳 400 Bad Request。

try-except：捕捉所有可能的例外，避免伺服器崩潰。

日誌記錄：使用 `logger.error()` 記錄錯誤，方便後續排查問題。

一致的回應格式：成功和失敗都使用 `{"success": bool, ...}` 的格式，方便前端處理。

日誌系統

系統使用 Python 的 logging 模組來記錄運行狀態：

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)

logger = logging.getLogger(__name__)

# 使用
```

```
logger.info("頁面處理成功")
logger.warning("速率限制，重試中")
logger.error("API 呼叫失敗")
```

日誌等級：

- **DEBUG**：詳細的除錯資訊
- **INFO**：一般的運行資訊
- **WARNING**：警告，但不影響運行
- **ERROR**：錯誤，可能影響功能

如何擴展

如果你想要擴展這個系統，以下是一些方向：

新增 AI 服務商

1. 建立新的生成器類別：

```
class OpenAIGenerator(GeneratorBase):
    def __init__(self, config):
        super().__init__(config)
        # OpenAI 初始化...

    def remove_text_from_image(self, image_base64):
        # OpenAI 的圖片編輯 API...
        pass

    def ocr_extract_text(self, image_base64):
        # OpenAI 的 OCR...
        pass
```

2. 在工廠中註冊：

```
class GeneratorFactory:
    @staticmethod
    def create(provider_type, config):
        if provider_type == 'gemini':
            return GeminiGenerator(config)
```

```
        elif provider_type == 'openai':
            return OpenAIGenerator(config)
```

3. 在 `providers.yaml` 中加入配置：

```
providers:
  openai:
    type: openai
    api_key: "YOUR_OPENAI_KEY"
    model: gpt-4-vision
```

新增 API 端點

```
@process_bp.route('/new-feature', methods=['POST'])
def new_feature():
    """新功能的 API"""
    data = request.get_json()
    # 實現邏輯...
    return jsonify({"success": True, "result": ...})
```

本章小結

本章介紹了簡報編修系統的後端架構：

Flask 框架：輕量靈活的 Python Web 框架，適合 API 開發

路由設計：使用 Blueprint 組織 API，處理 HTTP 請求

AI 生成器架構：使用抽象基類和工廠模式，支援多種 AI 服務商

Gemini API 整合：文字移除、背景修復、OCR 功能的實現

SSE 技術：即時回報處理進度

錯誤處理：輸入驗證、例外處理、日誌記錄

這個架構的設計讓系統：

- 易於維護和擴展

- 可以輕鬆新增其他 AI 服務商
 - 有良好的錯誤處理和日誌記錄
-

恭喜你完成了本書的所有內容！

從 NotebookLM 的核心功能，到五大神級簡報風格，到十五種資料視覺化圖表，到四種後製工具，到自動化系統，再到技術架構的深入解析——你已經掌握了 NotebookLM 簡報製作的完整知識體系。

希望這本書能幫助你更有效率地製作專業簡報，在工作和學習中發揮 NotebookLM 的強大潛力。