PopMart 監控系統開發設計圖

版本: 2.0.0
日期: 2025-07-01
作者: Manus AI

目錄

1. 系統概述

PopMart 監控系統是一個專門用於監控 PopMart 香港官網產品庫存和價格變化的工具。系統通過定期訪問 PopMart 官方 API，獲取最新的產品信息，並將其與本地數據庫中的歷史記錄進行比較，以識別價格變化、庫存狀態更新以及新產品上架等事件。

1.1 系統目標

1. **實時監控**：定期自動檢查 PopMart 官網的產品更新。

2. **數據追蹤**：記錄產品價格和庫存的歷史變化。
3. **特定產品關注**：允許用戶指定特定的產品系列或關鍵字進行重點監控。
4. **用戶友好界面**：提供直觀的 Web 界面，方便用戶查看監控結果。
5. **高效穩定**：在不影響 PopMart 官網正常運行的前提下，高效穩定地獲取數據。

1.2 系統功能

1. **產品數據獲取**：從 PopMart API 獲取產品列表、詳情和庫存信息。
2. **數據存儲**：將獲取的產品數據存儲到本地數據庫。
3. **變化檢測**：識別產品價格、庫存和其他屬性的變化。
4. **定時任務**：支持設置定時任務，自動執行數據更新。
5. **用戶界面**：提供 Web 界面，展示產品列表、變化歷史和監控狀態。
6. **過濾和搜索**：支持按品牌、系列、庫存狀態等條件過濾產品。

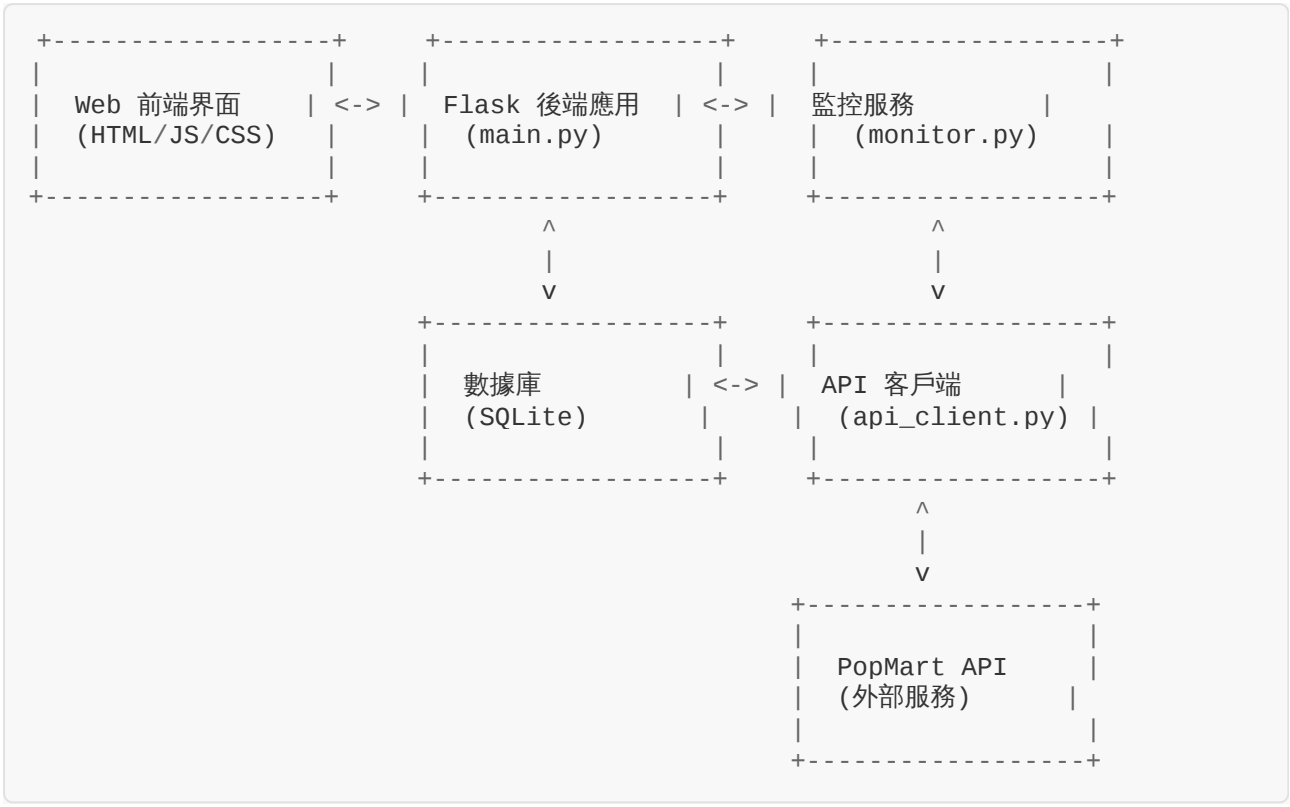
1.3 技術棧

- **後端**：Python 3.8+, Flask, SQLAlchemy, aiohttp
- **前端**：HTML, CSS, JavaScript (原生)
- **數據庫**：SQLite
- **部署**：Docker (可選)

2. 系統架構

本系統沿用並優化了原有的模組化架構，主要由以下幾個核心組件構成：

2.1 架構概覽



2.2 組件說明

1. **Web 前端界面**：提供用戶交互界面，展示產品列表、監控狀態和歷史記錄。使用原生 HTML、CSS 和 JavaScript 實現，保持輕量級和高性能。
2. **Flask 後端應用**：作為系統的核心控制器，處理來自前端的請求，協調各個服務模組的工作，並提供 RESTful API 接口。
3. **監控服務**：負責協調產品數據的獲取、處理和存儲流程，包括變化檢測和通知生成。
4. **API 客戶端**：封裝了與 PopMart 官方 API 的交互邏輯，處理 HTTP 請求、響應解析和錯誤處理。
5. **數據庫**：使用 SQLite 存儲產品數據、價格歷史和庫存歷史，支持高效的查詢和分析。
6. **排程器**：管理定時任務，按照設定的時間間隔自動觸發數據更新流程。

2.3 數據流

1. **數據獲取流程**：
2. 排程器觸發監控服務執行更新任務

3. 監控服務調用 API 客戶端獲取最新產品數據
4. API 客戶端發送請求到 PopMart API 並解析響應
5. 監控服務將獲取的數據與數據庫中的歷史記錄比較
6. 監控服務將新數據和變化記錄保存到數據庫
7. **用戶交互流程**：
8. 用戶通過 Web 界面發送請求
9. Flask 後端接收請求並調用相應的服務
10. 服務處理請求並返回結果
11. Flask 後端將結果返回給前端
12. 前端更新界面展示結果

3. 數據模型設計

數據模型是系統的核心，它定義了如何存儲和組織從 PopMart API 獲取的數據。本系統使用 SQLAlchemy ORM 框架來管理數據模型，主要包含以下幾個模型：

3.1 Product 模型

`Product` 模型用於存儲產品的基本信息和當前狀態。

```python
class Product(db.Model):
    """產品數據模型"""
    __tablename__ = 'products'

    id = db.Column(db.String(255), primary_key=True)
    name = db.Column(db.String(255), nullable=False)
    description = db.Column(db.Text)
    price = db.Column(db.Float, nullable=False)
    currency = db.Column(db.String(10), default='HKD')
    original_price = db.Column(db.Float)
    discount_price = db.Column(db.Float)
    image_url = db.Column(db.String(512))
    image_urls = db.Column(db.Text)  # JSON string
    video_url = db.Column(db.String(512))
    product_url = db.Column(db.String(512))
    category_id = db.Column(db.String(255))
    category_name = db.Column(db.String(255))
    brand_id = db.Column(db.String(255))
    brand_name = db.Column(db.String(255))
    series = db.Column(db.String(255))
    in_stock = db.Column(db.Boolean, default=False)
    stock_quantity = db.Column(db.Integer)
    max_purchase_quantity = db.Column(db.Integer)
    is_new = db.Column(db.Boolean, default=False)
    is_limited = db.Column(db.Boolean, default=False)
    is_pre_order = db.Column(db.Boolean, default=False)
    is_blind_box = db.Column(db.Boolean, default=False)
    release_date = db.Column(db.String(50))
    pre_order_start = db.Column(db.String(50))
    pre_order_end = db.Column(db.String(50))
    dimensions = db.Column(db.Text)  # JSON string
    weight = db.Column(db.Float)
    material = db.Column(db.String(255))
    tags = db.Column(db.Text)  # JSON string
    sku = db.Column(db.String(255))
    barcode = db.Column(db.String(255))
    view_count = db.Column(db.Integer, default=0)
    like_count = db.Column(db.Integer, default=0)
    review_count = db.Column(db.Integer, default=0)
    average_rating = db.Column(db.Float, default=0.0)
    created_at = db.Column(db.String(50), default=lambda:
datetime.now().isoformat())
    updated_at = db.Column(db.String(50))
    last_checked = db.Column(db.String(50))
```

3.2 PriceHistory 模型

`PriceHistory` 模型用於記錄產品價格的歷史變化。

```
class PriceHistory(db.Model):
    """價格歷史數據模型"""
    __tablename__ = 'price_history'

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    product_id = db.Column(db.String(255), db.ForeignKey('products.id'),
nullable=False)
    price = db.Column(db.Float, nullable=False)
    discount_price = db.Column(db.Float)
    timestamp = db.Column(db.String(50), default=lambda:
datetime.now().isoformat())
```

3.3 StockHistory 模型

`StockHistory` 模型用於記錄產品庫存狀態的歷史變化。

```
class StockHistory(db.Model):
    """庫存歷史數據模型"""
    __tablename__ = 'stock_history'

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    product_id = db.Column(db.String(255), db.ForeignKey('products.id'),
nullable=False)
    in_stock = db.Column(db.Boolean, nullable=False)
    stock_quantity = db.Column(db.Integer)
    timestamp = db.Column(db.String(50), default=lambda:
datetime.now().isoformat())
```

3.4 數據模型關係

- `Product` 與 `PriceHistory` 是一對多關係，一個產品可以有多條價格歷史記錄。
- `Product` 與 `StockHistory` 是一對多關係，一個產品可以有多條庫存歷史記錄。

3.5 數據模型設計考量

1. **使用 JSON 字符串**：對於複雜的數據結構（如 `image_urls`、`dimensions` 和 `tags`），使用 JSON 字符串格式存儲，以便在 SQLite 中保存結構化數據。
2. **時間戳格式**：使用 ISO 格式的字符串存儲時間戳，而不是數據庫的日期時間類型，以確保跨平台兼容性和易於序列化。
3. **自動生成時間戳**：`created_at`、`updated_at` 和 `timestamp` 字段使用 lambda 函數自動生成當前時間的 ISO 格式字符串。

4. **外鍵關係**：使用外鍵確保數據完整性，例如 `PriceHistory` 和 `StockHistory` 中的 `product_id` 引用 `Product` 的 `id`。

4. API 客戶端設計（`popmart_api_client.py`）

API 客戶端是系統與 PopMart 官方 API 交互的橋樑，負責發送 HTTP 請求、處理響應和錯誤處理。本系統使用 `aiohttp` 庫實現異步 HTTP 請求，以提高性能和效率。

4.1 PopmartProduct 數據類

`PopmartProduct` 是一個數據類，用於表示從 API 獲取的產品數據。

```python
@dataclass
class PopmartProduct:
    """Popmart商品數據結構"""
    id: str
    name: str
    price: float
    currency: str = "HKD"
    original_price: Optional[float] = None
    discount_price: Optional[float] = None
    image_url: str = ""
    image_urls: List[str] = None
    video_url: Optional[str] = None
    product_url: str = ""
    category_id: Optional[str] = None
    category_name: Optional[str] = None
    brand_id: Optional[str] = None
    brand_name: Optional[str] = None
    series: Optional[str] = None
    in_stock: bool = True
    stock_quantity: Optional[int] = None
    max_purchase_quantity: Optional[int] = None
    is_new: bool = False
    is_limited: bool = False
    is_pre_order: bool = False
    is_blind_box: bool = False
    release_date: Optional[str] = None
    pre_order_start: Optional[str] = None
    pre_order_end: Optional[str] = None
    dimensions: Optional[Dict[str, float]] = None
    weight: Optional[float] = None
    material: Optional[str] = None
    tags: List[str] = None
    sku: Optional[str] = None
    barcode: Optional[str] = None
    view_count: int = 0
    like_count: int = 0
    review_count: int = 0
    average_rating: float = 0.0
    description: Optional[str] = None
    created_at: Optional[str] = None
    updated_at: Optional[str] = None
    last_checked: str = ""
```

4.2 PopmartAPIClient 類

`PopmartAPIClient` 類封裝了與 PopMart API 的所有交互邏輯。

```python
class PopmartAPIClient:
    """Popmart API客戶端"""

    def __init__(self, region: str = "hk"):
        self.region = region
        self.base_url = "https://prod-intl-api.popmart.com"
        self.web_base_url = f"https://www.popmart.com/{region}"
        self.session: Optional[aiohttp.ClientSession] = None
        self.rate_limiter = asyncio.Semaphore(3)  # 限制並發請求

        # 用戶代理池
        self.user_agents = [
            'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36',
            'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36',
            'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:121.0) Gecko/20100101
Firefox/121.0',
            'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
AppleWebKit/605.1.15 (KHTML, like Gecko) Version/17.1 Safari/605.1.15',
            'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/120.0.0.0 Safari/537.36',
            'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/120.0.0.0 Edge/120.0.0.0'
        ]
```

4.3 主要方法

1. **會話管理**:
2. `start_session()`: 啟動 HTTP 會話
3. `close_session()`: 關閉 HTTP 會話
4. `__aenter__()` 和 `__aexit__()`: 支持異步上下文管理器
5. **HTTP 請求**:
6. `_make_request()`: 發送 HTTP 請求並處理響應
7. `_get_base_headers()`: 獲取基礎請求頭
8. **產品數據獲取**:
9. `get_products()`: 獲取產品列表
10. `get_product_details()`: 獲取單個產品詳情
11. `search_products()`: 搜索產品
12. `check_inventory()`: 檢查產品庫存

13. `get_new_arrivals()`: 獲取新品
14. `get_limited_products()`: 獲取限量產品
15. **數據解析**:
16. `_parse_product_data()`: 解析產品數據

4.4 反爬蟲策略

為了避免觸發 PopMart 官網的反爬蟲機制，API 客戶端實現了以下策略：

1. **隨機用戶代理**：從用戶代理池中隨機選擇一個用戶代理。
2. **請求延遲**：每次請求前添加隨機延遲（1.5-4.0 秒）。
3. **並發限制**：使用信號量限制並發請求數量（最多 3 個）。
4. **錯誤處理**：對於 429（請求過多）錯誤，自動等待一段時間後重試。

4.5 錯誤處理

API 客戶端對不同類型的錯誤進行了處理：

1. **HTTP 狀態碼錯誤**:
2. 200: 成功，返回解析後的 JSON 數據
3. 429: 請求過多，等待後重試
4. 403: 禁止訪問，可能觸發了反爬蟲機制
5. 404: 資源不存在
6. 其他: 記錄錯誤並返回 None
7. **網絡錯誤**:
8. 超時錯誤: 記錄錯誤並返回 None
9. 其他異常: 記錄錯誤並返回 None

5. 監控服務設計 (`monitor.py` 和 `specific_monsters_scraper.py`)

監控服務是系統的核心業務邏輯層，負責協調產品數據的獲取、處理和存儲流程，以及變化檢測和通知生成。

5.1 SpecificMonstersScraper 類

`SpecificMonstersScraper` 類專門用於獲取特定系列的產品數據。

```
class SpecificMonstersScraper:
    """特定產品爬蟲服務"""

    def __init__(self, api_client: PopmartAPIClient):
        self.api_client = api_client
        # 預設監控的產品名稱列表
        self.default_product_names = [
            "SKULLPANDA 溫度系列",
            "MOLLY 幻想大亨系列",
            "DIMOO 迷失在太空系列",
            "LABUBU 太空旅行系列",
            "PUCKY 森林精靈系列",
            "HIRONO 夢幻星球系列"
        ]
        self.product_name_to_id_map: Dict[str, str] = {}
```

5.2 MonitorService 類

`MonitorService` 類是監控服務的主要實現，負責協調整個監控流程。

```
class MonitorService:
    """監控服務"""

    def __init__(self, api_client: PopmartAPIClient):
        self.api_client = api_client
        self.scraper = SpecificMonstersScraper(api_client)
        self.update_progress = {"status": "idle", "percentage": 0, "message":
""}
        self._running = False
```

5.3 主要方法

1. **數據更新**:
2. `update_products()`: 協調產品數據的更新流程

3. `_process_products()`：處理獲取的產品列表，包括保存和變化檢測
4. **數據轉換**：
5. `_create_product_from_api()`：從 API 產品創建數據庫產品對象
6. `_update_product_from_api()`：從 API 產品更新數據庫產品對象
7. **狀態管理**：
8. `get_update_progress()`：獲取更新進度
9. `is_updating()`：檢查是否正在更新
10. **監控產品管理**：
11. `add_product_to_monitor()`：添加產品到監控列表
12. `remove_product_from_monitor()`：從監控列表中移除產品
13. `get_monitored_products()`：獲取當前監控的產品列表

5.4 更新流程

監控服務的更新流程包括以下步驟：

1. **獲取新品**：從 API 獲取最新上架的產品。
2. **獲取限量商品**：從 API 獲取限量版產品。
3. **獲取特定監控產品**：通過 `SpecificMonstersScraper` 獲取特定系列的產品。
4. **搜索關鍵字產品**：如果提供了關鍵字，搜索相關產品。
5. **處理產品數據**：對於每個獲取的產品，進行以下處理：
6. 檢查產品是否已存在於數據庫中
7. 如果存在，更新產品信息
8. 如果不存在，創建新產品記錄
9. 檢測價格變化，如有變化則記錄到價格歷史
10. 檢測庫存變化，如有變化則記錄到庫存歷史
11. 對新品和限量商品進行特殊處理

5.5 進度跟踪

監控服務使用 `update_progress` 字典來跟踪更新進度，包括以下字段：

- `status`：更新狀態，可能的值包括 "idle"、"running"、"completed" 和 "failed"
- `percentage`：完成百分比，範圍為 0-100
- `message`：當前步驟的描述信息

前端可以通過定期查詢 `/api/update_progress` 端點來獲取更新進度。

6. 排程器設計（`scheduler.py`）

排程器負責管理定時任務，按照設定的時間間隔自動觸發數據更新流程。

6.1 Scheduler 類

`Scheduler` 類是排程器的主要實現，使用 Python 的 `threading` 和 `asyncio` 模塊來管理異步任務。

```
class Scheduler:
    """排程器服務"""

    def __init__(self, monitor_service: MonitorService):
        self.monitor_service = monitor_service
        self._loop = None
        self._thread = None
        self._running = False
        self._interval = 300  # 默認5分鐘
        self._keywords = []
```

6.2 主要方法

1. **任務管理**：
2. `start()`：啟動排程器，定期執行更新任務
3. `stop()`：停止排程器
4. `is_running()`：檢查排程器是否正在運行
5. `get_status()`：獲取排程器狀態
6. `update_settings()`：更新排程器設置

7. **內部方法**：
8. `_run_loop()`：在獨立線程中運行異步事件循環
9. `_schedule_updates()`：異步調度更新任務

6.3 工作原理

排程器的工作原理如下：

1. **啟動排程器**：
2. 創建新的事件循環
3. 在獨立線程中運行事件循環
4. 設置更新間隔和關鍵字
5. **定期執行更新**：
6. 在事件循環中，每隔指定的時間間隔執行一次更新任務
7. 如果上一次更新任務尚未完成，則跳過本次更新
8. 使用 `monitor_service.update_products()` 執行更新任務
9. **停止排程器**：
10. 設置 `_running` 標誌為 False
11. 安全地停止事件循環
12. 等待線程結束

6.4 線程安全

排程器使用以下策略確保線程安全：

1. **獨立線程**：在獨立的線程中運行事件循環，避免阻塞主線程。
2. **標誌控制**：使用 `_running` 標誌控制排程器的運行狀態。
3. **優雅關閉**：在停止排程器時，安全地關閉事件循環並等待線程結束。

7. Flask 後端應用設計（`main.py`，`routes/monitor.py`，`extensions.py`）

Flask 後端應用是整個 PopMart 監控系統的門戶，它負責處理來自前端的所有 HTTP 請求，協調各個服務模組（API 客戶端、監控服務、數據存儲），並將處理結果返回給前端。它將作為一個輕量級的 Web 服務器，提供 RESTful API 接口和靜態文件服務。

7.1 核心功能與職責

1. **HTTP 請求處理**：接收並解析來自前端的各種請求，例如觸發數據更新、獲取產品列表、查詢更新進度等。
2. **路由管理**：定義清晰的 URL 路由，將不同的請求映射到對應的處理函數。
3. **服務協調**：作為中間層，調用 `MonitorService` 來執行數據更新，調用 `PopmartDataStorage` 來獲取數據，並將結果格式化後返回給前端。
4. **靜態文件服務**：提供 `index.html`、CSS 和 JavaScript 文件給瀏覽器。
5. **錯誤處理**：處理應用層的錯誤，並返回友好的錯誤響應。
6. **應用程式初始化**：初始化 Flask 應用、數據庫連接、服務實例和排程器。

7.2 模組設計

7.2.1 `main.py` (應用程式入口)

`main.py` 負責 Flask 應用程式的初始化、配置加載、數據庫連接設置以及各個服務實例的創建和注入。它還將啟動排程器和 Flask 服務器。

```python
def create_app():
    app = Flask(__name__, static_folder=os.path.join(os.path.dirname(__file__),
'static'))

    # 啟用CORS
    CORS(app)

    # 配置密鑰
    app.config['SECRET_KEY'] = 'popmart_monitor_secret_key'

    # 配置數據庫
    app.config['SQLALCHEMY_DATABASE_URI'] =
f"sqlite:///{os.path.join(os.path.dirname(__file__), 'database',
'popmart_data.db')}"
    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
    db.init_app(app)

    # 創建數據庫表
    with app.app_context():
        db.create_all()
        logger.info("數據庫表已創建或已存在。")

    # 初始化服務
    api_client = PopmartAPIClient(region="hk")

    # 啟動API客戶端會話
    @app.before_request
    def ensure_api_client_session():
        if not hasattr(g, 'api_client_session_started'):
            # 在第一個請求前啟動會話
            loop = asyncio.new_event_loop()
            asyncio.set_event_loop(loop)
            try:
                loop.run_until_complete(api_client.start_session())
                g.api_client_session_started = True
                logger.info("API客戶端會話已啟動")
            except Exception as e:
                logger.error(f"啟動API客戶端會話失敗: {e}")
            finally:
                loop.close()

    # 初始化其他服務
    monitor_service = MonitorService(api_client)
    scheduler = Scheduler(monitor_service)

    # 將服務實例注入到 Flask app
    app.api_client = api_client
    app.monitor_service = monitor_service
    app.scheduler = scheduler

    # 註冊藍圖
    app.register_blueprint(monitor_bp, url_prefix='/api')

    @app.route('/', defaults={'path': ''})
    @app.route('/<path:path>')
    def serve(path):
        static_folder_path = app.static_folder
        if static_folder_path is None:
            return "Static folder not configured", 404

        if path != "" and os.path.exists(os.path.join(static_folder_path,
```

```
path)):
            return send_from_directory(static_folder_path, path)
        else:
            index_path = os.path.join(static_folder_path, 'index.html')
            if os.path.exists(index_path):
                return send_from_directory(static_folder_path, 'index.html')
            else:
                return "index.html not found", 404

    return app
```

7.2.2 `routes/monitor.py` (監控相關路由)

此模組定義與監控功能相關的 API 端點，例如觸發更新、獲取產品列表和查詢更新進度。

```
def run_async(func):
    """裝飾器：運行異步函數"""
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)
        try:
            return loop.run_until_complete(func(*args, **kwargs))
        finally:
            loop.close()
    return wrapper

@monitor_bp.route('/update_products', methods=['POST'])
@run_async
async def update_products():
    """觸發產品數據更新"""
    monitor_service = current_app.monitor_service

    if monitor_service.is_updating():
        return jsonify({"status": "error", "message": "更新已在進行中，請勿重複觸
發。"}), 409

    # 從請求中獲取關鍵字，如果沒有則為空列表
    data = request.get_json()
    keywords = data.get('keywords', []) if data else []

    # 執行更新任務
    await monitor_service.update_products(keywords=keywords)

    return jsonify({"status": "success", "message": "產品更新任務已啟動。"})
```

7.3 API 端點設計

Flask 後端應用提供以下 API 端點：

1. **產品數據管理**：
2. `GET /api/products`：獲取產品列表，支持分頁和過濾

3. `GET /api/products/<product_id>`：獲取單個產品詳情，包括價格和庫存歷史
4. **更新任務管理**：
5. `POST /api/update_products`：觸發產品數據更新
6. `GET /api/update_progress`：獲取更新進度
7. **排程器管理**：
8. `GET /api/scheduler/status`：獲取排程器狀態
9. `POST /api/scheduler/start`：啟動排程器
10. `POST /api/scheduler/stop`：停止排程器
11. `PUT /api/scheduler/settings`：更新排程器設置
12. **監控產品管理**：
13. `GET /api/monitored_products`：獲取當前監控的產品列表
14. `POST /api/monitored_products`：添加產品到監控列表
15. `DELETE /api/monitored_products/<product_name>`：從監控列表中移除產品

7.4 異步處理

Flask 本身不直接支持異步處理，但本系統通過以下方式實現了異步功能：

1. **異步裝飾器**：使用 `run_async` 裝飾器將異步函數包裝為同步函數，以便在 Flask 路由中使用。
2. **獨立事件循環**：在獨立的事件循環中執行異步操作，避免阻塞主線程。
3. **會話管理**：在第一個請求前啟動 API 客戶端會話，並在應用關閉時安全地關閉會話。

8. 前端介面設計 (`src/static/index.html`)

前端介面是使用者與 PopMart 監控系統互動的視窗。為了保持系統的輕量級和快速開發的特性，我們採用原生的 HTML、CSS 和 JavaScript 來構建前端。其主要目標是提供一個直觀、響應迅速的介面，用於展示產品資訊、觸發數據更新以及顯示更新進度。

8.1 核心功能與職責

1. **產品列表展示**：清晰地列出所有監控的產品，包括產品名稱、價格、庫存狀態、圖片等關鍵資訊。
2. **手動更新觸發**：提供一個按鈕，允許使用者手動觸發產品數據的更新。
3. **更新進度顯示**：通過進度條和文字提示，實時顯示後台數據更新的狀態和進度。
4. **響應式設計**：確保介面在不同尺寸的設備上（桌面、平板、手機）都能良好地顯示和操作。
5. **錯誤提示**：當後台操作失敗時，能夠向使用者提供友好的錯誤提示。

8.2 頁面結構

前端頁面主要包含以下幾個部分：

1. **控制面板**：包含手動更新按鈕、更新進度條和狀態消息。
2. **過濾選項**：允許用戶按庫存狀態、商品類型、品牌等條件過濾產品。
3. **產品列表**：以網格形式展示產品卡片，每個卡片包含產品圖片、名稱、價格和庫存狀態等信息。
4. **排程器設置**：允許用戶設置自動更新的時間間隔和關鍵字，並啟動或停止排程器。
5. **監控產品管理**：顯示當前監控的產品列表，並允許添加或移除產品。

8.3 JavaScript 邏輯

前端 JavaScript 邏輯主要包括以下幾個部分：

1. **數據獲取**：使用 `fetch` API 從後端獲取產品列表、更新進度等數據。
2. **DOM 操作**：動態創建和更新產品卡片、進度條等 DOM 元素。
3. **事件處理**：處理按鈕點擊、表單提交等用戶交互事件。
4. **定時輪詢**：定期查詢更新進度和排程器狀態。
5. **錯誤處理**：捕獲並處理 AJAX 請求的錯誤，向用戶顯示友好的錯誤提示。

8.4 CSS 樣式

CSS 樣式使用了響應式設計原則，確保頁面在不同設備上都能良好地顯示：

1. **網格佈局**：使用 CSS Grid 佈局產品卡片，自動適應不同屏幕寬度。
2. **彈性盒子**：使用 Flexbox 佈局控制面板和過濾選項，實現靈活的排列。
3. **媒體查詢**：使用媒體查詢在小屏幕設備上調整佈局和元素大小。
4. **過渡效果**：添加適當的過渡效果，提升用戶體驗。

9. 錯誤處理與日誌記錄

一個健壯的系統必須具備完善的錯誤處理和日誌記錄機制。這不僅有助於在開發和調試階段快速定位問題，也能在生產環境中監控系統運行狀況，及時發現和解決潛在的故障。本系統在各個層面實施統一的錯誤處理策略和詳細的日誌記錄。

9.1 錯誤處理策略

錯誤處理遵循以下原則：

1. **分層處理**：
    - **API 客戶端層 (`popmart_api_client.py`)**：負責處理與外部 API 互動時的低級錯誤，如網路連接問題、HTTP 狀態碼錯誤（403, 404, 429, 500 等）和 API 返回的業務錯誤碼。此層應盡可能地捕獲異常，並將其轉換為更高級別的、對上層服務友好的錯誤信息或返回 `None`，避免直接拋出原始異常。
    - **業務邏輯層 (`monitor.py`, `scheduler.py`)**：負責處理業務邏輯中的錯誤，例如數據解析失敗、數據庫操作異常、產品 ID 無法匹配等。此層應根據錯誤類型決定是重試、跳過還是終止當前操作，並向上層報告處理結果。
    - **Flask 後端應用層 (`main.py`, `routes/monitor.py`)**：負責處理來自前端的請求驗證錯誤、路由錯誤以及從業務邏輯層傳遞上來的異常。此層應向前端返回標準化的錯誤響應（例如 JSON 格式的錯誤信息和適當的 HTTP 狀態碼），並記錄詳細的錯誤日誌。
2. **自動重試**：對於暫時性的錯誤（如網路瞬斷、API 429 頻率限制），系統實施帶有指數退避策略的自動重試機制，以提高操作的成功率。
3. **錯誤隔離**：確保單個模組或單個任務的失敗不會導致整個系統崩潰。例如，即使某個產品的數據獲取失敗，也不應影響其他產品的更新。
4. **用戶友好提示**：對於前端用戶，錯誤信息應簡潔明了，避免顯示技術細節，並引導用戶採取下一步行動（例如「請稍後重試」或「聯繫管理員」）。

9.2 日誌記錄

系統使用 Python 的 `logging` 模組進行統一的日誌記錄，並配置不同的日誌級別，以便於監控和調試。

1. **日誌級別**：
    - `DEBUG`：用於開發和調試階段，記錄詳細的程式執行流程、變數值等信息。
    - `INFO`：記錄系統正常運行時的關鍵事件，如任務啟動/完成、數據更新成功、API 調用成功等。
    - `WARNING`：記錄可能導致問題但尚未影響系統正常運行的事件，如 API 返回非預期數據、部分產品數據獲取失敗等。
    - `ERROR`：記錄導致功能失敗的錯誤，如 API 請求失敗（非 429）、數據庫操作失敗、關鍵業務邏輯異常等。
    - `CRITICAL`：記錄導致系統無法繼續運行的嚴重錯誤，如應用程式啟動失敗、核心服務崩潰等。
2. **日誌內容**：
    - **時間戳**：精確到毫秒，方便追溯事件發生時間。
    - **日誌級別**：明確標識日誌的嚴重程度。
    - **模組/函數名**：指明日誌來源，方便定位程式碼位置。
    - **消息內容**：清晰描述事件或錯誤的詳細信息。
    - **異常信息**：對於錯誤和警告，應包含完整的異常堆棧信息 (`exc_info=True`)，以便於問題分析。
    - **關鍵數據**：在必要時，記錄與事件相關的關鍵數據（例如產品 ID、API 響應狀態碼等），但應避免記錄敏感信息。

9.3 錯誤排查流程

當系統出現問題時，將遵循以下排查流程：

1. **檢查前端提示**：查看前端介面是否有任何錯誤消息或進度異常。
2. **查看後端日誌**：檢查 Flask 應用程式的控制台輸出或日誌文件，從 `ERROR` 和 `WARNING` 級別的日誌開始，向上追溯問題發生的源頭。

3. **隔離問題模組**：根據日誌信息，判斷問題可能發生在哪個模組（例如 API 客戶端、監控服務、數據存儲）。
4. **重現問題**：如果可能，嘗試在開發環境中重現問題，以便進行更詳細的調試。
5. **單元測試/集成測試**：針對問題模組編寫或運行相關的測試用例，驗證其行為是否符合預期。
6. **逐步調試**：使用調試工具（如 `pdb`）逐步執行程式碼，檢查變數值和執行流程。
7. **API 響應檢查**：如果問題與 API 互動有關，檢查實際的 API 請求和響應內容，與預期的 API 文檔進行比對。

10. 部署與維護

系統的部署和後續維護是確保其長期穩定運行的關鍵環節。本節將概述 PopMart 監控系統的部署建議和日常維護的最佳實踐。

10.1 部署建議

對於此類 Python Flask 應用程式，有幾種常見的部署方式，考慮到其輕量級和監控的特性，我們推薦以下方案：

10.1.1 本地部署 (開發/測試環境)

- **環境準備**：
  - 安裝 Python 3.8+。
  - 安裝 `pip` (Python 包管理器)。
- **依賴安裝**：
  - 導航到專案根目錄。
  - 運行 `pip install -r requirements.txt` 安裝所有必要的 Python 庫。
- **啟動應用**：
  - 運行 `python main.py` 啟動 Flask 開發服務器。
  - 應用程式將在 `http://0.0.0.0:5000` 上運行。
- **數據庫**：SQLite 數據庫文件 (`popmart_data.db`) 將自動在專案根目錄下創建。

10.1.2 生產環境部署 (推薦使用 Docker)

為了確保環境的一致性、簡化部署流程和提高可擴展性，強烈建議使用 Docker 進行生產環境部署。

- **Docker 容器化**：
    - **Dockerfile**：創建一個 `Dockerfile` 來定義應用程式的運行環境、依賴和啟動命令。這將確保應用程式在任何支持 Docker 的環境中都能以相同的方式運行。
    - **Docker Compose**：如果未來系統需要集成其他服務（如 Redis 用於進度緩存、PostgreSQL 用於數據庫），可以使用 `docker-compose.yml` 文件來定義和管理多個服務的部署，簡化服務間的協調。
- **部署流程 (Docker)**：
    1. **構建 Docker 映像**：在專案根目錄下運行 `docker build -t popmart-monitor .`。
    2. **運行 Docker 容器**：運行 `docker run -p 5000:5000 popmart-monitor`。這將把容器內部的 5000 端口映射到主機的 5000 端口。
    3. **持久化數據**：為了防止容器重啟或刪除時數據丟失，應將 SQLite 數據庫文件映射到主機的持久化存儲卷：`docker run -p 5000:5000 -v /path/on/host/data:/app/popmart_data.db popmart-monitor`。
- **Web 服務器**：在生產環境中，不應直接使用 Flask 自帶的開發服務器。應使用生產級的 WSGI 服務器（如 Gunicorn 或 uWSGI）來運行 Flask 應用，並通過 Nginx 或 Apache 作為反向代理，處理靜態文件和負載均衡。

10.2 日常維護

系統部署後，需要進行持續的監控和維護，以確保其穩定、高效運行。

1. **日誌監控**：
    - 定期檢查應用程式日誌，特別是 `ERROR` 和 `WARNING` 級別的日誌，及時發現和處理異常。
    - 可以集成日誌收集工具（如 ELK Stack 或 Grafana Loki），集中管理和分析日誌。
2. **性能監控**：

- 監控 CPU、記憶體和網路使用情況，確保系統資源充足。
- 監控 API 請求的響應時間和成功率，及時發現 API 服務的潛在問題。

3. **數據庫備份**：
   - 定期備份 `popmart_data.db` 文件，防止數據丟失。對於 SQLite，可以直接複製文件進行備份。
4. **依賴更新**：
   - 定期檢查並更新 `requirements.txt` 中列出的 Python 庫，以獲取最新的功能、性能優化和安全補丁。
   - 更新後，應在測試環境中充分測試，確保沒有引入新的問題。
5. **API 變化適應**：
   - PopMart 官方 API 可能會不定期更新。應定期檢查其官網或開發者文檔，了解 API 的最新變化。
   - 如果 API 結構發生變化，需要及時更新 `popmart_api_client.py` 和相關的數據解析邏輯。
6. **反爬蟲策略調整**：
   - 如果系統頻繁遇到 403 或 429 錯誤，可能需要調整 `popmart_api_client.py` 中的反爬蟲策略，例如增加延遲、擴展用戶代理池、引入代理 IP 等。

11. 常見問題與解決方案

在系統開發和運行過程中，可能會遇到各種問題。以下是一些常見問題及其解決方案：

11.1 API 相關問題

1. **問題**：API 請求頻繁返回 429 (Too Many Requests) 錯誤。 **解決方案**：
2. 增加請求之間的延遲時間，例如從 1.5-4.0 秒增加到 3.0-8.0 秒。
3. 減少並發請求數量，例如將信號量從 3 降低到 1。
4. 實施更複雜的重試策略，例如指數退避。
5. **問題**：API 請求返回 403 (Forbidden) 錯誤。 **解決方案**：

6. 檢查請求頭是否包含必要的信息，例如 User-Agent、Accept 等。
7. 擴展用戶代理池，使用更多的真實瀏覽器 User-Agent。
8. 考慮使用代理 IP 輪換請求來源。
9. **問題**：API 響應結構發生變化，導致解析失敗。 **解決方案**：
10. 更新 `_parse_product_data()` 方法以適應新的響應結構。
11. 添加更健壯的錯誤處理，例如使用 `get()` 方法獲取字典值，並提供默認值。
12. 實施版本檢測機制，根據 API 版本選擇不同的解析邏輯。

11.2 數據庫相關問題

1. **問題**：數據庫文件損壞或無法訪問。 **解決方案**：
2. 從最近的備份恢復數據庫文件。
3. 如果沒有備份，創建新的數據庫文件，並重新獲取產品數據。
4. 實施定期備份機制，防止類似問題再次發生。
5. **問題**：數據庫查詢性能下降。 **解決方案**：
6. 為常用查詢添加索引，例如 `Product.name`、`Product.brand_name` 等。
7. 優化查詢語句，避免使用 `LIKE` 操作符進行全文搜索。
8. 定期清理歷史數據，例如刪除過舊的價格和庫存歷史記錄。

11.3 異步相關問題

1. **問題**：異步操作阻塞主線程，導致 Web 界面無響應。 **解決方案**：
2. 確保所有耗時操作都在獨立的事件循環中執行。
3. 使用 `run_async` 裝飾器包裝異步函數，避免阻塞 Flask 請求處理。
4. 考慮使用消息隊列（如 Celery）來處理長時間運行的任務。
5. **問題**：異步操作未正確關閉，導致資源洩漏。 **解決方案**：
6. 確保在應用關閉時正確關閉所有異步資源，例如 `aiohttp.ClientSession`。
7. 使用 `atexit` 模塊註冊清理函數，確保在應用退出時執行清理操作。

8. 使用上下文管理器（`async with`）來管理異步資源的生命週期。

11.4 前端相關問題

1. **問題**：前端頁面加載緩慢，特別是產品列表。 **解決方案**：
2. 實施分頁加載，每次只加載一部分產品。
3. 優化圖片加載，使用懶加載技術。
4. 減少 DOM 操作，使用文檔片段（`DocumentFragment`）批量更新 DOM。
5. **問題**：前端與後端通信失敗。 **解決方案**：
6. 檢查 CORS 配置，確保允許前端域名訪問後端 API。
7. 添加更健壯的錯誤處理，在請求失敗時提供友好的錯誤提示。
8. 實施重試機制，在網絡不穩定時自動重試失敗的請求。

12. 未來擴展方向

隨著系統的使用和需求的變化，可能需要進行功能擴展和性能優化。以下是一些潛在的擴展方向：

12.1 功能擴展

1. **通知系統**：
2. 添加電子郵件通知功能，當監控的產品價格下降或庫存狀態變化時發送通知。
3. 實現 WebSocket 或 Server-Sent Events，提供實時通知。
4. 添加移動端推送通知，方便用戶隨時了解產品變化。
5. **數據分析**：
6. 添加價格趨勢分析功能，展示產品價格的歷史變化趨勢。
7. 實現庫存預測功能，基於歷史數據預測產品何時會缺貨或補貨。
8. 添加熱門產品分析，幫助用戶發現受歡迎的產品。
9. **用戶系統**：

10. 添加用戶註冊和登錄功能，允許多用戶使用系統。
11. 實現用戶偏好設置，允許用戶自定義監控產品和通知方式。
12. 添加用戶權限管理，區分普通用戶和管理員。

12.2 性能優化

1. **數據庫優化**：
2. 遷移到更強大的數據庫系統，如 PostgreSQL 或 MySQL，以支持更大規模的數據和更複雜的查詢。
3. 實施數據分區策略，將歷史數據和當前數據分開存儲，提高查詢性能。
4. 添加緩存層，減少對數據庫的直接訪問。
5. **API 客戶端優化**：
6. 實現更智能的重試策略，根據不同類型的錯誤採取不同的重試方式。
7. 添加請求優先級機制，優先處理重要的請求。
8. 實現請求批處理，減少 API 調用次數。
9. **前端優化**：
10. 使用現代前端框架（如 React、Vue）重構前端，提供更好的用戶體驗。
11. 實現前端狀態管理，減少對後端的請求。
12. 添加客戶端緩存，減少重複數據的加載。

12.3 架構優化

1. **微服務架構**：
2. 將系統拆分為多個微服務，例如 API 客戶端服務、監控服務、通知服務等。
3. 使用消息隊列（如 RabbitMQ、Kafka）實現服務間的異步通信。
4. 實施服務發現和負載均衡，提高系統的可擴展性和可靠性。
5. **容器化和編排**：
6. 使用 Docker Compose 或 Kubernetes 管理容器化的服務。
7. 實施自動擴展策略，根據負載自動調整服務實例數量。

8. 添加健康檢查和自動恢復機制，提高系統的可用性。
9. **監控和日誌**：
10. 集成專業的監控工具（如 Prometheus、Grafana），實時監控系統性能和健康狀況。
11. 實施分布式追踪（如 Jaeger、Zipkin），跟踪請求在各個服務間的流轉。
12. 集成日誌聚合工具（如 ELK Stack、Graylog），集中管理和分析日誌。

通過這些擴展和優化，PopMart 監控系統可以不斷發展，滿足更多用戶的需求，並在更大規模的數據和更高的並發下保持穩定運行。