



FastAPI 200 422 解釋



為什麼fast api swagger ui會顯示200、422



在使用 FastAPI 開發應用程式時，Swagger UI 通常會根據你的路由定義和模型設置，自動產生 API 文檔，並顯示可能的 HTTP 狀態碼。`200` 和 `422` 是最常見的兩個狀態碼，以下是它們的意義與可能原因：

1. HTTP 200

- **意義：**請求成功，伺服器返回的響應是預期的結果。
- **原因：**
 - 當你在 FastAPI 中定義路由並設置正確的返回模型時，默認情況下，FastAPI 假設成功的響應將返回 `200` 狀態碼。
 - 例如：

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id, "name": "Test Item"}
```

Swagger UI 中會顯示 `200`，表示成功時的狀態。

2. HTTP 422

- **意義：**請求語法正確，但語義錯誤，伺服器無法處理。
- **原因：**
 - FastAPI 使用 **Pydantic** 進行請求數據的驗證。如果請求中的參數或資料格式不符合模型定義，就會返回 `422` 狀態碼。
 - 例如：
 - 路由中定義的參數類型與實際請求中提供的不匹配。
 - 請求的 JSON 或表單數據缺少必需字段或格式不正確。

示例代碼：



```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    price: float

@app.post("/items/")
async def create_item(item: Item):
    return {"name": item.name, "price": item.price}
```

- 如果請求時缺少 `name` 或 `price`，或者 `price` 無法轉換為浮點數（例如發送 `abc`），FastAPI 會返回 `422 Unprocessable Entity`。

Swagger UI 顯示的原因

FastAPI 的 OpenAPI 文檔（Swagger UI）會根據你的程式定義，自動生成所有可能的狀態碼，並在每個路由下展示。例如：

- `200`：成功響應。
- `422`：自動添加，表示資料驗證失敗的可能性。

你無需特別指定這些狀態碼，FastAPI 會基於你的路由邏輯自動生成對應的文檔。

如何處理 `422`？

1. 確保請求資料與 Pydantic 模型一致。
2. 驗證路由參數類型是否正確。
3. 如果需要自定義錯誤響應，可以使用自定義異常處理器：

```
from fastapi import FastAPI, HTTPException
from fastapi.responses import JSONResponse

app = FastAPI()

@app.exception_handler(422)
async def validation_exception_handler(request, exc):
    return JSONResponse(
        status_code=422,
        content={"message": "Validation error", "details": exc.errors()},
    )
```

這樣可以自定義返回的錯誤訊息格式。



如果有更多細節或範例想討論，可以提供程式碼，我可以幫你分析！



我的ubuntu 22.04環境的mariadb異常，怎麼重新安裝



在 Ubuntu 22.04 上，如果 MariaDB 出現異常，重新安裝可以幫助解決問題。以下是詳細的步驟，涵蓋卸載、清理和重新安裝 MariaDB 的過程。

步驟 1：停止 MariaDB 服務

在卸載之前，先停止 MariaDB 服務以防止可能的衝突。

```
sudo systemctl stop mariadb
```

步驟 2：卸載 MariaDB

卸載 MariaDB 相關套件和依賴項。

```
sudo apt purge mariadb-server mariadb-client mariadb-common mariadb-backup mariadb-core-*
```

步驟 3：清理剩餘的文件

刪除 MariaDB 的數據庫文件和配置文件，確保環境乾淨。

```
sudo rm -rf /etc/mysql /var/lib/mysql
sudo apt autoremove --purge -y
sudo apt autoclean
```

步驟 4：重新安裝 MariaDB

1. 更新軟體包清單：



```
sudo apt update
```

2. **安裝 MariaDB**：Ubuntu 22.04 默認軟體庫包含 MariaDB，執行以下命令重新安裝：

```
sudo apt install mariadb-server mariadb-client -y
```

3. **驗證安裝版本**（選擇性）：如果需要特定版本，可以手動添加 MariaDB 官方軟體庫：

```
sudo apt install software-properties-common
sudo apt-key adv --fetch-keys 'https://mariadb.org/mariadb_release_signing_key.asc'
sudo add-apt-repository 'deb [arch=amd64] http://mirror.aarnet.edu.au/pub/MariaDB/repo/10.6/ubuntu jammy main'
sudo apt update
sudo apt install mariadb-server mariadb-client
```

步驟 5：啟動 MariaDB

安裝完成後，啟動 MariaDB 服務並設置為開機自啟。

```
sudo systemctl start mariadb
sudo systemctl enable mariadb
```

步驟 6：執行安全設置

重新安裝後需要執行安全設置來配置 MariaDB。

```
sudo mysql_secure_installation
```

按照提示完成以下操作：

- 設置 root 密碼。
- 移除匿名用戶。
- 禁止 root 遠端登錄。
- 刪除測試數據庫。
- 重新加載權限表。

步驟 7：驗證 MariaDB 運行狀態

確認 MariaDB 是否正常運行。



```
sudo systemctl status mariadb
```

檢查 MariaDB 是否能正常連接：

```
mysql -u root -p
```

常見問題排查

- **問題 1：** MariaDB 無法啟動
查看日誌以獲取詳細錯誤信息：

```
sudo journalctl -u mariadb
```

- **問題 2：** 端口被占用
確保 3306 端口未被其他應用佔用：

```
sudo netstat -tulnp | grep 3306
```

如果還有其他具體問題，可以提供錯誤信息，我可以協助你進一步排查！



swagger ui/metrics HTTP/1.1 404Not found



在使用 FastAPI 或其他基於 ASGI 的框架（如 Starlette）開發時，Swagger UI 中的 `/metrics`` 路徑返回 `404 Not Found``，通常是因為該路徑未在應用中明確定義或相應的擴展未啟用。以下是解決這個問題的方法及可能的原因。

1. 確認是否需要 `/metrics`` 路徑

- **什麼是 `/metrics``？**
 - `/metrics`` 通常用於暴露應用程式的監控數據，供監控工具（如 Prometheus）使用。它並不是 FastAPI 或 Swagger UI 的內建路徑。
- **預期使用場景：**
 - 如果你希望應用提供 `/metrics`` 路徑，需明確定義該路徑的處理程式或使用相關擴展（例如 `prometheus-fastapi-instrumentator``）。



2. 使用 `prometheus-fastapi-instrumentator` 添加 `/metrics` 支援

如果目標是提供 Prometheus 格式的監控數據，請按以下步驟進行設置：

1. 安裝擴展

```
pip install prometheus-fastapi-instrumentator
```

2. 設置 FastAPI 應用 在你的應用程式中導入並初始化 `Instrumentator`：

```
from fastapi import FastAPI
from prometheus_fastapi_instrumentator import Instrumentator

app = FastAPI()

# 初始化 Instrumentator
Instrumentator().instrument(app).expose(app)
```

3. 啟動應用並訪問 `/metrics` 啟動應用後，你可以通過 `/metrics` 路徑查看監控數據。

3. 如果 `/metrics` 不是預期功能

- 如果你並不需要 `/metrics` 功能，但仍然有 `404` 問題，這可能與其他錯誤相關，例如錯誤配置或 URL 拼寫錯誤。
- 檢查是否有其他代碼中明確定義了 `/metrics`，或是否有第三方模組期望該路徑存在。

4. 排查問題

如果 `/metrics` 路徑應該存在但仍然返回 `404`，嘗試以下步驟：

1. 檢查路徑是否已註冊 列出所有註冊的路由，確認 `/metrics` 是否存在。

```
for route in app.routes:
    print(route.path)
```

2. 查看日誌 檢查應用日誌，看是否有與 `/metrics` 相關的錯誤信息。

```
journalctl -u <your-service> # 如果部署為服務
```

3. 檢查環境配置 確保環境中未被過濾或代理器（如 Nginx）誤攔截。





5. 如果問題仍然存在

提供更多細節，例如：

- 是否使用了第三方擴展或監控工具。
- 完整的應用程式配置（包括中間件或路由）。
- 部署環境（本地、Docker、Kubernetes）。

這樣我可以協助進一步排查！



`docker build -t`，在dockerfile :`RUN mysql_safe`後 build image 卡住



在 Dockerfile 中使用 `RUN mysql_safe` 後卡住的問題，通常是因為 `mysql_safe` 是一個長時間運行的進程，設計用於啟動並監控 MySQL/MariaDB 服務。這導致 `docker build` 無法完成，因為 `RUN` 指令預期短時間內完成命令執行，而非啟動持續運行的服務。

以下是詳細解釋和解決方法：

為什麼卡住？

1. `RUN` 指令的特性：

- `RUN` 用於執行構建時的命令，目的是在生成的鏡像中執行腳本或命令，並在命令執行後結束。
- `mysql_safe` 是一個持續運行的守護進程，`RUN mysql_safe` 永遠不會返回，導致構建卡住。

2. 應用程式啟動邏輯錯誤：

- `mysql_safe` 應該在容器運行時（`CMD` 或 `ENTRYPOINT`）啟動，而不是在構建時。

解決方案

方法 1：僅在運行時啟動 MySQL

將 `mysql_safe` 移至 `CMD` 或 `ENTRYPOINT`，讓它在容器啟動時執行，而非在構建時。

範例 Dockerfile：



```
# 使用官方 MariaDB 映像作為基礎
FROM mariadb:latest

# 複製需要的文件或執行其他構建指令
COPY ./my.cnf /etc/mysql/my.cnf

# 確保數據目錄已初始化
RUN mysqld --initialize-insecure

# 設置默認命令
CMD ["mysqld_safe"]
```

方法 2：使用多階段構建初始化 MySQL 數據

如果你需要在構建階段執行 MySQL 初始化腳本，可以使用以下方式：

1. 初始化數據到數據目錄：

```
FROM mariadb:latest

# 初始化數據庫
RUN mysqld --initialize-insecure --datadir=/var/lib/mysql

# 複製初始數據文件
COPY ./init.sql /docker-entrypoint-initdb.d/

# 設置默認命令
CMD ["mysqld_safe"]
```

注意：將初始化腳本（如 `init.sql`）放置於 `/docker-entrypoint-initdb.d/` 目錄下，MariaDB 或 MySQL 映像會在啟動時自動執行這些腳本。

方法 3：在構建時臨時啟動 MySQL

如果非要在構建階段啟動 MySQL，可以使用臨時啟動方式，確保命令在執行完畢後結束。

範例：

```
FROM mariadb:latest

# 複製初始化腳本
COPY ./init.sql /tmp/init.sql

# 臨時啟動 mysqld 並執行初始化腳本
RUN mysqld_safe --skip-networking & \
    sleep 5 && \
```




```
mysql -u root -e "source /tmp/init.sql" && \  
killall mysqld_safe
```

其他注意事項

1. **使用官方基礎映像：** 如果可能，盡量使用 MariaDB 或 MySQL 的官方映像，這些映像已處理好啟動和初始化邏輯。
2. **避免長時間進程在 `RUN` 指令執行：** `RUN` 應該僅用於執行短時命令，持續運行的應用程序應在容器啟動後執行。
3. **調試構建過程：** 在構建時添加調試指令，例如 `sleep` 或 `echo`，檢查問題發生的位置：

```
RUN mysqld_safe & sleep 10 && echo "MySQL started"
```

如果仍有問題，提供更詳細的 Dockerfile 或錯誤日誌，我可以幫助進一步排查！



await、asyncio.sleep(1)用法



在 Python 的異步編程中，`await` 和 `asyncio.sleep(1)` 是常見的用法，分別用於調用協程和模擬異步的延遲操作。以下是詳細說明及示例：

1. `await` 的作用

- **意義：** `await` 是用於調用協程（coroutine）的關鍵字。協程是可以暫停並在稍後繼續執行的函數，通常用於處理 I/O 操作或其他耗時任務。
- **用法：**
 - 只能在 `async` 定義的函數中使用。
 - 當執行 `await` 時，程序會暫停該協程的執行，讓出控制權，允許其他協程運行。
- **示例：**

```
import asyncio  
  
async def say_hello():  
    print("Hello")  
    await asyncio.sleep(1) # 模擬等待 1 秒  
    print("World")
```





```
asyncio.run(say_hello())
```

輸出：

```
Hello  
(等待 1 秒)  
World
```

2. `asyncio.sleep(1)` 的作用

- **意義：** `asyncio.sleep(1)` 是一個協程，會暫停當前協程的執行約 1 秒，允許事件循環 (event loop) 切換到其他協程。
- **對比 `time.sleep(1)`：**
 - `time.sleep(1)` 是阻塞操作，會停止整個程序的執行，無法讓其他協程運行。
 - `asyncio.sleep(1)` 是非阻塞操作，僅暫停當前協程，其它協程仍可運行。
- **示例：**

```
import asyncio  
  
async def task_one():  
    print("Task one starts")  
    await asyncio.sleep(2) # 等待 2 秒  
    print("Task one ends")  
  
async def task_two():  
    print("Task two starts")  
    await asyncio.sleep(1) # 等待 1 秒  
    print("Task two ends")  
  
async def main():  
    await asyncio.gather(task_one(), task_two())  
  
asyncio.run(main())
```

輸出：

```
Task one starts  
Task two starts  
Task two ends  
Task one ends
```

解釋：

- `task_one` 和 `task_two` 同時開始執行。



- 在 `task_two` 等待的 1 秒內，`task_one` 繼續執行。
- `task_two` 結束後，`task_one` 繼續執行剩餘的 1 秒。

3. `await` 和 `asyncio.sleep` 的結合

`await` 用於等待協程的完成，而 `asyncio.sleep` 是一個常見的協程，適合模擬非阻塞延遲。

- 應用場景：
 - 模擬異步操作的延遲（如網絡請求、I/O 操作）。
 - 測試異步代碼的性能。
 - 控制協程執行的節奏。

4. 使用注意事項

- `await` 只能在 `async` 函數內使用。
- 當需要並行運行多個協程時，使用 `asyncio.gather()` 或 `asyncio.create_task()`。

5. 高階示例：多任務協程的應用

```
import asyncio

async def download_file(file):
    print(f"Start downloading {file}")
    await asyncio.sleep(2) # 模擬下載時間
    print(f"Finished downloading {file}")

async def main():
    files = ["file1.txt", "file2.txt", "file3.txt"]
    tasks = [download_file(file) for file in files]
    await asyncio.gather(*tasks) # 同時執行所有下載任務

asyncio.run(main())
```

輸出：

```
Start downloading file1.txt
Start downloading file2.txt
Start downloading file3.txt
Finished downloading file1.txt
```



```
Finished downloading file2.txt  
Finished downloading file3.txt
```

如果還有其他具體應用場景或問題，隨時告訴我！