Efficient COM Port Programming with Python: A Practical Guide

# 引言

在這篇文章中，我們將探討一些多工處理的概念，並將它用在雙向通訊的情境中。這些原理不僅適用於Python或這個雙向通訊的例子，實際上，它們可以應用於任何程式語言中與任何情境中。希望通過知曉這些原理，能夠設計出更好的程式架構。

目錄

[引言 1](#_Toc148912676)

[主題討論 2](#_Toc148912677)

[阻塞 2](#_Toc148912678)

[多工 2](#_Toc148912679)

[線程與行程 4](#_Toc148912680)

[行程 4](#_Toc148912681)

[多行程 5](#_Toc148912682)

[線程 5](#_Toc148912683)

[多線程 5](#_Toc148912684)

[多線程 vs 多行程 5](#_Toc148912685)

[非同步與事件驅動 6](#_Toc148912686)

[範例演示 8](#_Toc148912687)

[8051端（組合語言） 8](#_Toc148912688)

[電腦端（PYTHON語言） 10](#_Toc148912689)

[結語 13](#_Toc148912690)

[參考資料 13](#_Toc148912691)

# 主題討論

為實現資料的雙向傳輸，我們需以多工處理方式設計程式。其他範例皆以多線程的方式來實現多工處理。在開始這個主題之前，筆者想先指出：吳奇隆學長在他的COM Port Programming with Python中關於Thread套件的註解「可邊設定輸出時邊等待輸入數值」需謹慎看待，因為牽涉到Python特有的GIL問題，關於這點我們稍後也會討論。而至於楊永楠學長的COM Port Programming with C# 和陳鍇瀚學長的COM Port Programming with Visual BASIC，其對於Thread的使用說明可能會引起混淆。由於軟體阻塞和多工處理的概念確實不容易理解，我們首先討論第一個主題——阻塞。

## 阻塞

為何Verilog的assignment會區分blocking與non-blocking？答案或許跟C語言考試經常出現的「快捷運算」有關。實際上，這都歸功於阻塞操作的影響。以開車做比喻，如果前車不走，後車就無法前進，這就是「阻塞」。

阻塞是序向邏輯電路與組合邏輯電路最明顯的差異之一，序向電路的輸出不只取決於當前輸入，也與前一次的狀態有關，因此需要區分先後順序。為了確保執行順序，就必須讓第二個操作必須等待第一個操作完成，這個「等待」的過程就是一種「阻塞」，序向電路通常使用脈波邊緣觸發來實現阻塞效果。

與硬體相反，軟體先天就是阻塞的。例如，當一行程式碼在執行完畢之前，程式不會跳到下一行，這種情況就源自於阻塞操作。因此，許多語言在邏輯運算中會使用「快捷運算」來改善執行速度（快捷運算表現出的行為與硬體邏輯閘不能說完全相似，只能說一模一樣）。

阻塞也可以視為一種「占用」，程式占用了CPU的線程，導致其他的程式需「等待」該程式釋放線程後才能執行。這正是楊永楠學長與陳鍇瀚學長提到的「使用While持續接收，會一直卡住」的原因。

## 多工

多工顧名思義就是處理多個工作，而

這種占用問題不僅在軟體中出現，無線通訊也因為共用介質（如空氣等）而引起占用問題。為了解決這類問題，無線通訊使用路復用作為解決方案，常見的方式包含以下四種：

1. 分時多工（TDM）：將時間切分成不同時段，輪流使用通訊媒介。
2. 分頻多工（FDM）：將頻率切分成不同頻段，同時使用通訊媒介。
3. 空間多工（SDM）：通過波束成形技術將訊號對準接收站發射。
4. 分碼多工（CDM）：在發送時添加代號，以便接收端識別特定數據。

回到我們的主題，我們不僅可以使用多線程實現多工處理，還能夠使用**分時多工**的方式！從微算機的角度來看，分時多工被稱為「半雙工」，而同時多工則自然就是「全雙工」。那麼要如何實現這個半雙工的模式呢？與其說出來不如看一下Python範例。

一張含有 文字, 螢幕擷取畫面, 字型 的圖片

自動產生的描述

（半雙工示例）

可以看出，我們透過分時多工的方法，成功避免了楊永楠學長與陳鍇瀚學長所遇到的卡住問題。不同嗎？確實如此，他們卡住的是GUI介面而非資料傳送。雖然不知道C# 以及Visual BASIC能否將函數註冊到GUI的迴圈之中，但在Python可以透過以下的方式來達成。

一張含有 文字, 螢幕擷取畫面, 軟體 的圖片

自動產生的描述

（搭配GUI介面的示例）

上方範例同樣使用了分時多工的方式，與先前不同的是這邊也將GUI加入至等待隊列。至於發送訊息的功能，只需將函數綁定到按鈕上即可，在此就略過不詳細說明。由這個範例可以得知：並行多工（Concurrency）不一定需要平行（Parallelism）。

## 線程與行程

在本節中，我們將討論線程（或稱作：執行緒，英文Thread）與行程（Process，大陸翻譯叫做進程）的差別。

### 行程

行程（Process）是執行中的程式實例，每一個行程有各自的ID，且互相獨立。百聞不如一見，直接上圖！在工作管理員中，可以看到筆者此時開啟的應用程式，而應用程式後面括號的數字就是該應用程式開啟了多少個實例，比方說Edge瀏覽器就開了38個實例。

一張含有 文字, 螢幕擷取畫面, 軟體, 數字 的圖片

自動產生的描述

當我們點進詳細資料，確實看到了Edge正在執行超級多的實例，而每個實例都有自己的PID（Process ID），所以Microsoft Edge (38) 表示Edge開啟了38個行程。

一張含有 文字, 螢幕擷取畫面, 軟體, 數字 的圖片

自動產生的描述

（可以看到Edge開了許多行程）

### 多行程

電腦同時有許多的行程需要執行，而作業系統通過上下文交換（Context Switch）的方式在不同的行程之間不斷切換，給使用者造成一種多個行程同時執行的假象。在搶占式多工的系統中，每一個行程將會輪流執行不定長度的時間。到執行時間到達限制時，作業系統會產生一個中斷，將行程執行期間使用的所有暫存器儲存起來，並安排下一個行程執行。因此同時打開太多應用程式，各個程式平均能分配到的資源就會下降，讓使用者感覺電腦變慢。

### 線程

線程（Thread）究竟是什麼呢？以蓋房子來比喻，應用程式向建築公司（作業系統）下訂單，作業系統安排不同的訂單，分配給工人（Thread）來施工。Thread是真正執行工作的實體，也是程式執行的最小單元。雙核心是指將兩個CPU核心集成在同一顆晶片上，因此搭載這顆晶片的電腦具有2個處理單元。然而，自從Intel在2002年推出超執行緒技術（Hyper-Threading），現今的一顆處理單元能夠模擬兩個線程，所以通常將CPU核心數乘以二就是線程的數量。不過Intel第12代後的Alder Lake架構CPU，其中混合了效能核心P core和效率核心E core，因此在12代以後的CPU上，這個方法不再適用。

### 多線程

Intel的超執行緒技術並不限制一個行程只能使用兩個線程。實際上，線程（Thread）分為三個不同的層次：

* Hardware Thread：物理上能夠同時運作的執行單元數量（例如i7-10510U有8個線程）
* Kernel Thread：由作業系統管理（OS）的thread，何時放到hardware thread執行由OS決定
* User Thread：由程式透過Library所建，OS 會將這些thread 綁定到 kernel thread 上

Windows作業系統的線程數量通常沒有限制，在N:1 的Threading Model當中，程式可以創建N個User Thread，但這些User Thread最終只會綁定到同一個Kernel Thread。（在Linux與macOS對線程的創建數量有限制，更多資訊可參考：[How Many Threads Can a Java VM Support? | Baeldung](https://www.baeldung.com/jvm-max-threads)）

不同的程式語言可能使用不同的線程模型，例如Go語言的goroutine設計讓Golang支援高效的平行處理。而Python與Java使用的是1:1 model，即每個程式執行的thread都對應到一個kernel thread。（註：早期的Java使用Green thread進行多線程處理，這些Green thread是在User Level層面進行管理。後來Java才將多線程執行轉移到native thread，即Kernel Thread。）

由於Python的預設直譯器CPython的內存管理不具備線程安全性，所以引入了全域直譯器鎖（Global Interpreter Lock，縮寫為GIL）來避免多個Kernel Thread 同時執行Python bytecode。儘管GIL在某種程度上提供了線程安全性，卻也導致Python在執行時限制只允許一個線程運行，這成為Python在多線程處理上常受批評的原因。

### 多線程 vs 多行程

最初我們期望透過平行計算（Parallel Computing）來充分發揮多核心處理器（例如i7-10510U就有4個核心）的性能，然而由於全域直譯器鎖的存在，Python的Multithreading往往無法達到我們預期的效果。在這種情況下，我們可以借助第三方套件如Numba或是Dask這類第三方套件。Numba不僅支援CPU上的平行運算，還能運行在NVIDIA CUDA和AMD ROCm等GPU環境上。如果曾使用Python進行機器學習（Machine Learning），可能聽說Dask，它過支援pandas、Numpy、scikit-learn、XGBoost等套件，Dask不但能在本地執行，也能擴展到叢集上，實現「分散式運算」（Distributed computing）。在本節中，我們將比較多線程與多行程的概念，並明白各自的優點和缺點以及如何在這兩方式中選擇。

如果想在原生Python中平行運算，可以使用內建的multiprocessing庫，透過建立行程（Process）的方式來讓Python以平行的方式執行。這種方式的原理是各行程的GIL是相互獨立的。在類Unix系統下，也可以使用os.fork() 達到相同的效果。（註：更多關於os.fork() 與multiprocessing的差異可以看看這篇討論：[Python multiprocessing process vs. standalone Python VM - Stack Overflow](https://stackoverflow.com/questions/2276117/python-multiprocessing-process-vs-standalone-python-vm)。）

由於CPython的GIL在進行阻塞 I/O操作時會釋放，因此對於I/O密集型任務，多數文章建議使用multithreading，而對於計算密集型的任務使用multiprocessing。而在沒有GIL的語言中，要使用Multithreading還是Multiprocessing？可以參考文章[Multithreading vs. Multiprocessing: What's the Difference? | Indeed.com](https://www.indeed.com/career-advice/career-development/multithreading-vs-multiprocessing)。和多行程類似，當kernel thread的數量大於hardware thread的數量時， 線程也會進行上下文交換。更多資訊可以看這篇文章：[Difference between Thread Context Switch and Process Context Switch - GeeksforGeeks](https://www.geeksforgeeks.org/difference-between-thread-context-switch-and-process-context-switch/)。

但使用multithreading的效能不如預期，也會造成除錯上的困難，因此multithreading已經逐漸被下個章節所介紹的非同步、事件驅動這類方式取代[[1]](#footnote-2)。

## 非同步與事件驅動

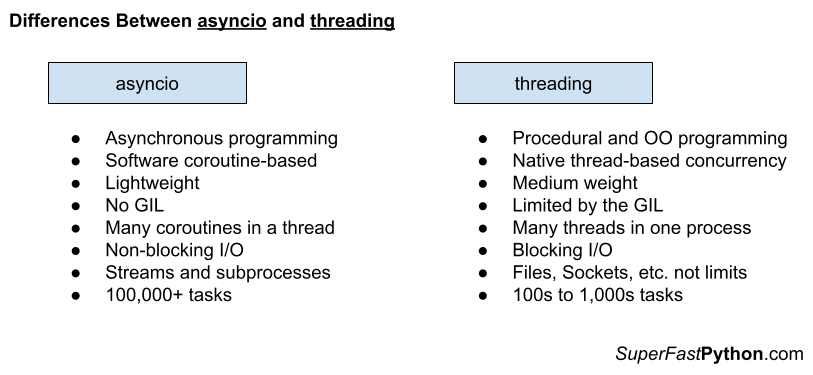
在本節中，我們將討論非同步與事件驅動的概念和影響。非同步與事件驅動是實現多工的另外兩種方式，它們可以提高程式的反應速度和靈活性。網路上有個例子可以幫助我們理解阻塞非阻塞與同步非同步的概念[[2]](#footnote-3)。

**老張燒開水的故事**（故事來源網路）。  
出場人物：老張，水壺兩把（普通水壺，簡稱水壺；會響的水壺，簡稱響壺）。  
  
老張愛喝茶，時常二話不說便煮起了開水。  
 **第一次**：老張置水壺於火上，立等水開（同步阻塞）。  
 > 水開了，但老張覺得自己這樣乾等有點傻。  
  
**第二次：**老張置水壺於火上，到客廳看電視，時不時看看廚房的水開了沒有（同步非阻塞）。  
 > 老張依舊覺得自己有點傻，於是買了把會響笛的水壺。水開之後，能發出嘀~~的噪音。  
  
**第三次：**老張置水壺於火上，立等水開。（非同步阻塞）  
 > 老張就這樣傻等壺響，覺得意義不大。  
  
**第四次：**老張把響水壺放在火上，去大廳看電視，直到響了再去拿壺。（異步非阻塞）  
 > 老張覺得自己聰明了。

同步與非同步的差異是在任務完成前需要持續關注該任務是否結束。會響的水壺代表一種事件通知的機制。而阻塞與非阻塞是在於等待的期間老張有沒有去做其他事情。

依據[Amdahl's law](https://en.wikipedia.org/wiki/Amdahl%27s_law)，就算可以無限提升平行化運算的速度，程式的總運算時間仍會被無法平行化運算加速的部分所侷限[[3]](#footnote-4)。[Multicore Scalability Through Asynchronous Work (vt.edu)](https://vtechworks.lib.vt.edu/handle/10919/104116) 提到以非同步的方式將不影響數據一致性的任務從關鍵路徑中移除，並移動到後台非同步執行，這樣可以減少關鍵路徑的大小，從而提高系統的性能和可擴展性。

需要注意的是Multithreading與Asyncio並不一樣，在有GIL的情況下，當Multithreading用於阻塞型的I/O任務時，GIL就會釋放，這時Python就會去執行其他線程。而Asyncio卻永遠都只有單個線程，且Asyncio只在處理非阻塞型的I/O上才有效果。計算密集型的任務無法使用。

[](https://superfastpython.com/asyncio-vs-threading/)

（Asyncio與Threading的比較[[4]](#footnote-5)）

我們有在Process的章節提及搶占式多工（Preemptive multitasking）的概念，而Asyncio是使用一種稱為協作式多工（Cooperative Multitasking）的方式。相較於搶佔式多工（Preemptive multitasking）由作業系統決定任務切換時機。協作式多工要求程式定時放棄（yield）自己的執行權，轉讓給下一個程式執行。這種透過程式之間合作而達到的多工方式，就稱作協作式多工。

適時放棄執行權也能提升程式整體的執行效率，非同步程式設計的技巧就是在進行阻塞操作時（也就是等待期間）將執行權轉交出去，在程式等待的同時，讓其他程式進行操作。若要觀看更多與非同步程式設計的討論可以閱讀「[程式設計該同步還是非同步？ | iThome](https://www.ithome.com.tw/voice/74544)」這篇文章。

下個章節將會演示如何在8051上實作非同步程式設計。期望讀者在明白軟硬體相似時，能有醍醐灌頂的感受。

**範例演示**

**範例功能**

這個示例演示了如何實現一個應用程式，允許電腦和8051通過UART介面互相傳輸8位元的數字。兩方將接收到的數字與本地數字相加，然後顯示結果。

一張含有 電子產品, 電路, 電子元件, 電路元件 的圖片

自動產生的描述一張含有 螢幕擷取畫面, 文字, 軟體, 數字 的圖片

自動產生的描述

（功能展示，左圖為8051、右圖為電腦端）

接線方式這裡就不多做說明，只提醒一點：作業繳交區有提供對應型號USB to UART Bridge的驅動程式，要安裝電腦才能夠找到COM Port。

## 8051端（組合語言）

範例將會展示三種不同的寫法。讀者可以比較這三者之間的差異。

一張含有 文字, 螢幕擷取畫面, 軟體, 多媒體軟體 的圖片

自動產生的描述

（8051阻塞範例程式片段）

上方的程式片段是來自其他的項目。放在這裡主要是讓讀者對阻塞的寫法有一定程度的了解。閱讀這個程式片段可以發現，當程式在得到輸入之前（RI等於1之前），程式會一直停（阻塞）在第七行。當然這只代表程式會等待輸入而已，不代表無法實現多工，依照不同的需求，可以設計成在輸出一個資料後等待輸入，並在輸入完成後進行輸出。如此往復也能稱作多工（雙向傳輸）。

一張含有 文字, 螢幕擷取畫面, 軟體 的圖片

自動產生的描述

（8051同步非阻塞範例）

而這個範例程式是同步非阻塞的寫法，以Java語言的角度來看就是NIO模型。程式以輪詢的方式不斷檢查（或者說等待）輸入或輸出事件，在事件發生之前我們的程式將持續運行，所以它是非阻塞的。以老張煮水的故事來說，老張（程式）會不斷的檢查（LOOP迴圈）開水有沒有滾（是否輸入RI/是否輸出TI）。由於我們將讀取輸入與進行輸出的工作交給底層，在這裡就是8051的硬體電路，因此我們不用處理輸入與輸出的部分。若是使用軟體，那I/O的部分就會交給外部進行處理，例如作業系統（Operating System，縮寫OS）或在網路程式設計上的話有可能是將資料丟給伺服器處理，直到伺服器處理完畢再傳送回來。（畢竟工作終究得要有人做）而在這段期間進行其他的任務就能稱為「非阻塞」。很明顯能知道，這個程式能夠處理輸入也能處理輸出，達到多工處理的功能（雙向傳輸）。

一張含有 文字, 螢幕擷取畫面, 軟體 的圖片

自動產生的描述

（8051異步非阻塞範例）

這個程式以Java來看就是AIO模型，程式碼使用中斷的方式（事件驅動）處理UART通訊。它設置了UART接收中斷（RI）的中斷服務程式UART\_ISR。當RI為1時，將觸發中斷並呼叫UART\_ISR處理接收，輸出同理。以老張煮水的故事來說，我們發送訊息（煮水）後並沒有等待TI變成1，而是可以進行其他的事情（雖然這邊是直接停在第20行）。而我們並沒有不斷的檢查水是否已經煮好（以LOOP檢查TI），而是讓水壺好的時候發出響聲（觸發中斷），這個方法似乎又比剛才的方式聰明了。

在Linux可以使用Epoll的方式來監聽多個描述符的狀態，當其中一個描述符變更時通知程式。而8051的RI與TI已經OR在一起，所以不能將輸入與輸出拆成兩個事件，實屬可惜。

## 電腦端（PYTHON語言）

在這個範例功能中，8051會持續不斷的傳送資料，而我們以Python語言開發的客戶端也將不停的接收資料（當然你也能讓8051只在資料改變時才傳送資料）。很典型的I/O密集行任務，範例是使用Python的Thread套件。由於GIL在線程進行阻塞式的I/O操作與呼叫time.sleep() 時都會釋放，所以這個Python實際上的運作模式是不斷的在GUI與讀取8051資料的線程之間切換（Context Switch）。

from serial import \*  
from serial.tools import list\_ports  
from tkinter import ttk  
from tkinter.messagebox import \*  
import tkinter as tk  
import threading  
import time  
  
baudrate = 9600  
  
  
# ----------------------------------------------- #  
ser: Serial = None  
  
def loop() -> None:  
 while(True):  
 # Update port list  
 port\_chose['values'] = [x.device for x in list\_ports.comports()]  
   
 if(ser is None or not ser.isOpen()):  
 continue  
  
 try:  
 data = ser.read(1)  
 ser.reset\_input\_buffer()  
 print(data)  
 text\_a.set(f'{int.from\_bytes(data)}')  
 entry\_a.update()  
 time.sleep(0.1)  
 except UnicodeDecodeError:  
 # print("UnicodeDecodeError")  
 pass  
   
 ans['text'] = str(int(text\_a.get()) + int(text\_b.get()))  
   
  
def connect():  
 global ser  
 if ser is not None:  
 ser.close()  
 del ser  
   
 if com\_port.get() == "":  
 showerror("Connecting Error", "Please select a port to connect")  
 return  
 ser = Serial(com\_port.get(), baudrate, timeout=0, writeTimeout=0)  
  
def send() -> None:  
 if ser is None or not ser.isOpen():  
 showerror("Sending Error", "Please connect to a port first")  
 return  
 ser.write(int(entry\_b.get()).to\_bytes())  
  
# --------------------- GUI --------------------- #  
root = tk.Tk()  
root.title("UART")  
root.geometry('360x300')  
root.minsize(360, 300)  
  
ctrl\_frame = tk.Frame(root)  
ctrl\_frame.pack(side=tk.TOP, fill=tk.X)  
  
com\_port = tk.StringVar()  
port\_chose = ttk.Combobox(ctrl\_frame, textvariable=com\_port) # 下拉式清單  
port\_chose.pack(side=tk.LEFT, padx=5, pady=3)  
  
connect\_btn = tk.Button(ctrl\_frame, text="Connect", command=connect)  
connect\_btn.pack(side=tk.LEFT)  
  
frame1 = tk.Frame(root)  
frame1.pack(side=tk.TOP, fill=tk.BOTH, padx=30, pady=30)  
  
text\_a = tk.StringVar()  
entry\_a = tk.Entry(frame1, width=4, font=('Arial', 30), state=tk.DISABLED,   
 textvariable=text\_a, justify=tk.RIGHT)  
entry\_a.pack(side=tk.LEFT, padx=10)  
  
tk.Label(frame1, text=" + ", font=('Arial', 40)).pack(side=tk.LEFT)  
  
text\_b = tk.StringVar()  
entry\_b = tk.Spinbox(frame1, width=4, font=('Arial', 30), textvariable=text\_b,   
 justify=tk.RIGHT, from\_=0, to=255, command=send)  
entry\_b.pack(side=tk.LEFT, padx=10)  
  
frame2 = tk.Frame(root)  
frame2.pack(side=tk.TOP, fill=tk.BOTH, padx=30, pady=30)  
  
tk.Label(frame2, text=" = ", font=('Arial', 40)).pack(side=tk.LEFT)  
ans = tk.Label(frame2, text="0", font=('Arial', 40))  
ans.pack(side=tk.LEFT)  
  
  
# --------------------- Main --------------------- #  
t = threading.Thread(target=loop)  
t.daemon = True  
t.start()  
  
root.mainloop()

**結語**

我們已經在這個雙向通訊的例子透過Python和8051的組合語言示例來說明阻塞與非同步等概念。這個例子當中也有能再進一步的地方，比如說讓8051只在資料改變時才傳送資料，或是讓Python的介面也使用非同步的寫法。筆者當初聽到RI/TI就聯想到程式設計的非同步程式設計，原本短短41行的程式碼居然能想到5000字的內容。希望這篇文章能給讀者其他範例沒有提到的內容，並提供一個全新角度的切入點來觀看其他的範例。

**參考資料**

* Mathew, A. (2020). Multicore scalability through asynchronous work (Master’s thesis). Virginia Polytechnic Institute and State University.
* Jess, A.(2017, April 18). Grok the GIL: How to write fast and thread-safe Python. Opensource.com. <https://opensource.com/article/17/4/grok-gil>
* Sysprog.(2023, August). 並行程式設計: 概念. Hackmd.io. <https://hackmd.io/@sysprog/concurrency/%2F%40sysprog%2Fconcurrency-concepts>
* 王建興.(2012, June 22). 程式設計該同步還是非同步？Ithome.com.tw. <https://www.ithome.com.tw/voice/74544>
* Jason Brownlee. (2023, January 20). Asyncio vs Threading in Python. <https://superfastpython.com/asyncio-vs-threading/>
* Pi314. (2016, Sep 19). Threading Models. Github.com. <https://github.com/pi314/pi314.notes/blob/main/os-threading-models.rst>
* Jack. (2021, July 15). Thread Model 介紹. Github.io. <https://yu-jack.github.io/2021/07/15/thread-model/>
* Yovan Li. (2019, February 10)OS Process & Thread (user/kernel) 筆記. Medium.com. <https://medium.com/@yovan/os-process-thread-user-kernel-筆記-aa6e04d35002>
* 跨元探索. (2022, August 4)【多工技術】OFDM 多重載波調變技術. Vocus.cc. <https://vocus.cc/article/62eb1500fd8978000182a421>
* Alwareadit. (2020, May 25). Difference between Thread Context Switch and Process Context Switch. Geeksforgeeks.org. <https://www.geeksforgeeks.org/difference-between-thread-context-switch-and-process-context-switch/>
* Indeed. (2023, February 4). Multithreading vs. Multiprocessing: What's the Difference? <https://www.indeed.com/career-advice/career-development/multithreading-vs-multiprocessing>
* Flaviu Cicio. (2023, September 21). How Many Threads Can a Java VM Support? Baeldung.com. <https://www.baeldung.com/jvm-max-threads>

1. 曾涉獵. (2015, May 17). 為什麼 thread (執行緒、線程)越少越好?. Goodspeedlee.blogspot.com. https://goodspeedlee.blogspot.com/2015/05/thread.html [↑](#footnote-ref-2)
2. Hejianhui. (2020, October 10 ).Netty学习前基本知识 — BIO 、NIO 、AIO 总结. Weixin.qq.com. https://mp.weixin.qq.com/s/7DrH3vdl0xVJp97Q-fjTAA [↑](#footnote-ref-3)
3. https://blog.maxkit.com.tw/2023/04/amdahls-law.html [↑](#footnote-ref-4)
4. https://superfastpython.com/asyncio-vs-threading/ [↑](#footnote-ref-5)