

技術者們
BONUS-B

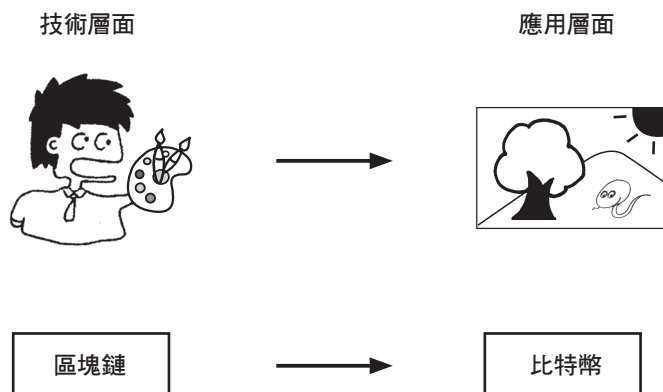
區塊鏈實作技術

- B-0 本章重點
- B-1 區塊鏈簡介 (Blockchain)
- B-2 Python 區塊鏈實作
- B-3 用 Flask 架設區塊鏈網站伺服器
- B-4 共識演算法
- B-5 實戰：運行區塊鏈 -
建立 2 個節點 (村民)



談到**區塊鏈 (Blockchain)**，不外乎先想到近年沸沸揚揚的虛擬貨幣-**比特幣 (Bitcoin)**，不少人也產生例如「區塊鏈 = 比特幣？」的疑問。

事實上區塊鏈是種**資訊技術**，而比特幣是基於區塊鏈技術發展的**虛擬貨幣應用**，它們之間的關係就像是繪畫技巧 (技術層面) 與風景畫 (應用層面)。



而區塊鏈的應用當然不只有比特幣，也有其他的新興虛擬貨幣如**以太幣 (Ether)**、**狗狗幣 (Dogecoin)** 或是**智慧合約 (Smart Contract)**…的應用，未來也被期待應用於**物聯網 (internet of things, IOT)**，解決資安痛點。

B-0 本章重點

本章將透過 Python 來建立自己的區塊鏈，設計自己虛擬貨幣的應用，如挖礦、交易…，並以 Python 的 Web 開發框架 **Flask** 來架設網站以提供一個讓使用者存取、操作區塊鏈的 **HTTP API** 介面，最後透過應用程式 **Postman** 來對我們架設的區塊鏈網站發出 HTTP 請求，進行上述的**挖礦、交易**…等等的操作。



TIPS Flask 是一個用 Python 撰寫的輕量級架站框架，可以用來進行 Web 網站開發，且具備高度的開發彈性。

本章將學到：

- 對區塊鏈的認識
- 用 Python 建立區塊鏈, 打造自己的虛擬貨幣應用
- 區塊鏈的加密技術: 雜湊函數
- 區塊鏈的挖礦演算法
- 區塊鏈的共識演算法
- 透過 Flask 框架開發 Web 網站

B-1 區塊鏈簡介 (Blockchain)

區塊鏈是種分散式帳本技術 (Distributed Ledger Technology, DLT) 的概念, 將資料分散儲存在每個參與者的電腦中, 若以比特幣來說, 區塊鏈就是一本記錄著過去比特幣歷史交易的**公共帳本**, 且每個用戶 (節點) 都會擁有此帳本。這樣的好處是打造更公開、透明的機制。

要實現比特幣、以太幣…等虛擬貨幣的應用, 事實上代表著以區塊鏈技術打造某個特定的「**網路**」, 而在此網路上流通的貨幣即為虛擬貨幣 (加密貨幣)。例如**比特幣網路**上即使用比特幣、**以太坊 (Ethereum) 網路**則使用以太幣。使用者加入這些網路即可依據各網路定義的規則來使用虛擬貨幣。

這些概念我們接下來舉個「**比特村**」的故事來說明, 讓您更容易理解。

B-1-0 比特村的故事

有個與世隔絕的村落叫做比特村, 在這個村莊中, 村長扮演著銀行的角色, 村民們可以到村長家去進行開戶、存錢的動作, 而村長會將每個村民戶頭中有多少錢, 或是交易…等, 紀錄在一本帳本上。

當村民之間要進行金錢交易時，例如：小美要向大華購買一件 699 元的牛仔褲，需要到村長那提領出 699 元，然後再交給大華換取牛仔褲，而大華收到 699 元後，再把錢拿給村長，請他存入自己的戶頭中。或者小美也可以寫一張交易單：「把 699 元從小美的戶頭轉到大華的戶頭」並交給村長紀錄在帳本上。



這樣的模式稱為「**中心化架構**」，而這也是現在銀行的運作機制，但這樣的機制有以下幾個隱憂：

- 1 信任問題：**村長必須很正派，不會胡亂更改帳本資料來謀取利益。
- 2 安全問題：**若村長家著火，帳本不小心被燒掉的話，大家的財產就會大亂。

有一天，村中有位叫中本聰的人，對於這樣中心化架構非常擔憂，於是將比特村中的實體貨幣換成等價的虛擬貨幣，並使用了「**區塊鏈**」的技術來打造一個新的**分散式架構**的交易管理機制。

這個機制是每個村民現在都可以在網路上建立自己的帳戶，村民之間若要進行交易，只需要在網路上發起一筆**端對端 (Peer-to-Peer)** 的交易 (匯款人：小美，收款人：大華，匯款金額：699 元)，這樣交易就完成了，中間不再需要村長這個中介人。

而這筆交易會傳遍整個比特村，也就是說所有人都會得知小美匯了 699 元給大華，全村村民作為公證人，雙方都無法耍賴。村中所有村民的交易紀錄都會同步登記在**公共帳本**上，而每個帳戶都會擁有此帳本，若有新的交易產生後也都會記錄在帳本上，而這本帳本其實就是我們所說的**區塊鏈**：



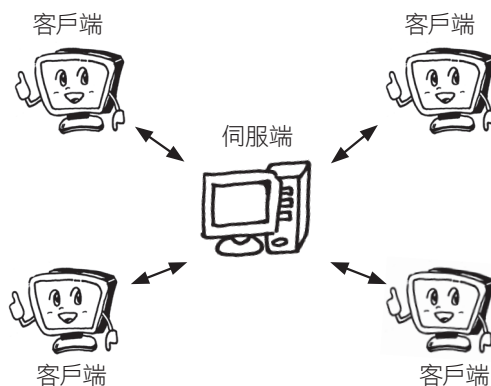
這樣的分散式架構解決了原本中心化管理的種種問題：

- 1 解決信任問題：**不再需要村長這個中介人，每個人都有相同的公共帳本，所以如果有心人士想竄改帳本資料的難度會很高，因為必須去竄改每一個村民手中的公共帳本。
- 2 解決安全問題：**若有人帳本遺失了，只要向其他村民索取帳本即可，因為大家都是一樣的帳本。

大概理解區塊鏈概念後，接下來我們來介紹幾個構成區塊鏈的要素。

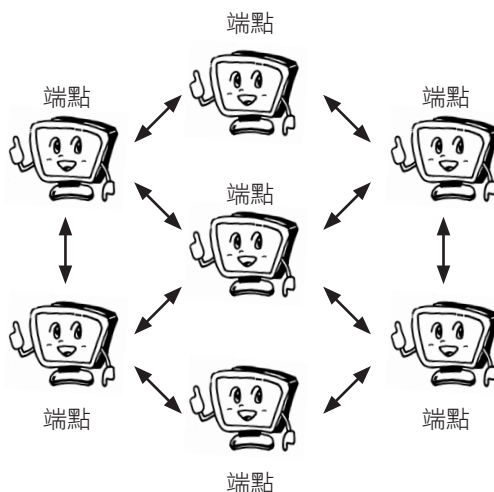
B-1-1 罷免村長銀行：去中心化形成的 P2P 網路架構

原先在比特村所運行的中心化架構（村長銀行）在資訊技術中稱為主從式網路架構（Client-Server），在這樣的網路架構之下會分成伺服器端（Server）與客戶端（Client）的存在，可以將村長想像成 Server、村民們就是 Client：



資料都儲存在伺服器端，而客戶端要索取資料、或者客戶端之間要進行溝通都得藉由伺服器端作為橋梁，所以當伺服器端發生問題時，這個網路架構就會崩潰。

而中本聰使用的區塊鏈則是 **P2P (Peer-to-Peer)** 的網路架構，這是一種端對端的平等架構，去除了中心化的管理（村長），每個端點（村民）都可以與鄰近的端點進行直接溝通；距離較遠的兩個端點也可以透過他們之間的端點來間接溝通：



這就是運行在比特村的區塊鏈網路，村民們都是網路中的用戶（端點）。

B-1-2 區塊鏈網路的參與者：村民

身為區塊鏈網路的參與者，村民可以依據行為而分成兩種角色（當然也可以同時扮演）：

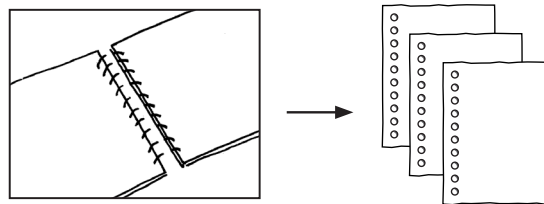
- 1 使用者 (User)：**發起交易，進行虛擬貨幣的匯款。
- 2 礦工 (Miner)：**挖掘新區塊 (Block)，賺取虛擬貨幣。

使用者的身分很容易理解，就像在比特村中，小美向大華購買牛仔褲，所以需要匯款 699 的虛擬貨幣到大華的戶頭中；而要理解為什麼需要挖礦產出新區塊，賺取虛擬貨幣的行為，就得從什麼是區塊開始說起了。

B-1-3 空白帳本紙：區塊 (block)

我們說比特村中，小美匯款給大華的這筆交易紀錄，會寫在每個人的公共帳本（區塊鏈）上然後同步更新，而這個公共帳本到底是怎麼來的呢？

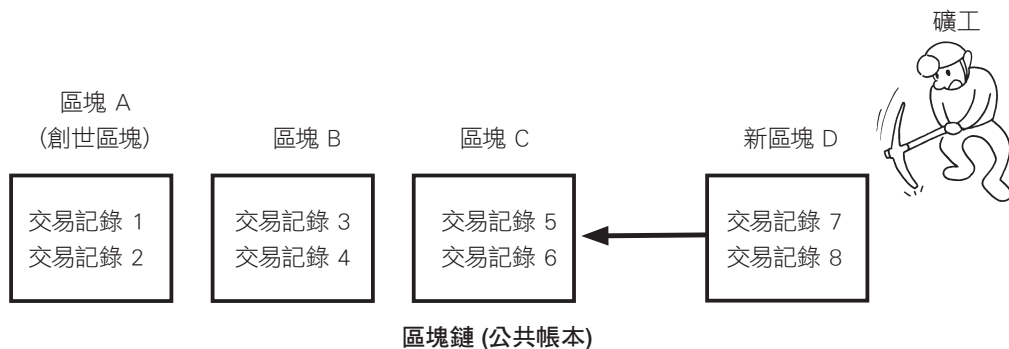
現在請將這本帳本想像成是那種打孔式活頁筆記本，可以壓開線圈，加入新的空白紙：



這些空白紙就是所謂的**區塊 (Block)**，而村民的交易紀錄都會寫在這些空白區塊上，寫滿後就夾進這本公共帳本，但這些區塊怎麼來的呢？

村民 (礦工) 必須透過電腦來計算一個數學問題，此動作即為**挖礦 (Mining)**：算出來的礦工即可產生出一個**新的區塊** (產出區塊的礦工會獲得虛擬貨幣的獎勵)，而帳本的第一個區塊也稱為**創世區塊 (Genesis Block)**

完成計算的礦工會將比特村村民們的交易資料 (例如小美匯款 699 給大華) 寫在新區塊上，並貼在區塊鏈的**最尾端**，一塊一塊的區塊前後連接，形成一條長長的鏈，這就是我們的**公共帳本：區塊鏈**。



稍後會實作一個**挖礦演算法**來讓礦工們計算這個數學問題 (找出某個數字，此數字與稍後會介紹的雜湊函數有關係)，而計算這個數學問題需要耗費一定的時間 (依據電腦的計算能力)，這也是區塊鏈難以竄改的原因之一 (詳細說明可參考本電子書最後的補充學習 A)。

B-1-4 公共帳本：區塊鏈

而區塊鏈中的區塊除了記錄交易資訊以外，還會記錄許多其他資訊，例如還會記錄前面一個區塊的雜湊值 (Hash Value)，透過這樣的方式來前後串接區塊，而也會透過雜湊值來確保區塊鏈中的區塊沒有被替換掉。



TIPS 雜湊值是區塊鏈使用的加密技術之一，稍後會加以說明。

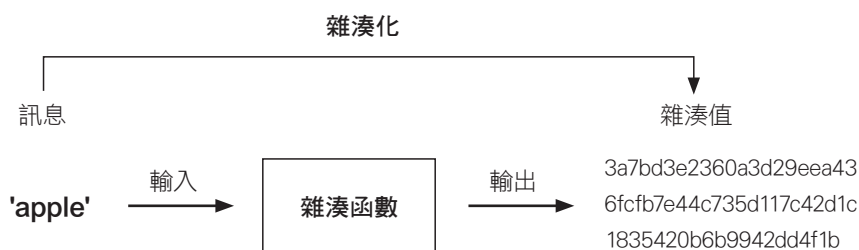
而透過 P2P 網路架構，這本帳本會分散在每個村民手中 (儲存在每個用戶的電腦中)，稍後也會透過 Python 實作共識演算法，透過此演算法來確保每個村民手中的帳本都是相同的。

B-2 Python 區塊鏈實作

接下來我們要使用 Python 來建立一個區塊鏈網路，建立虛擬貨幣應用的第一步，首先我們要先介紹區塊鏈中的一個加密技術 - 雜湊函數：

B-2-0 訊息的指紋：加密技術-雜湊函數

建構區塊鏈需要用到加密技術：把需要加密的訊息透過雜湊函數 (Hash Function) 轉換成雜湊值 (Hash Value)，這樣的過程可稱為雜湊化，如下圖：



稍後透過 Python 實作區塊鏈時，在產生區塊或是進行挖礦演算法，都會使用到雜湊函數。而雜湊函數有多種類型，在 Python 中可以匯入內建模組 **hashlib** 來使用各種不同的雜湊函數，並透過以下屬性查看有哪些類型的雜湊函數：


```
import hashlib
print('可用的雜湊函數:', hashlib.algorithms_available)
    輸出 {'md4', 'md5', 'sha256', 'RIPEMD160',...}
print('跨平台可用的雜湊函數:', hashlib.algorithms_guaranteed)
    輸出 {'md5', 'sha256',...}
```

不同的雜湊函數會輸出不同長度的雜湊值。例如 **SHA-256** 與 **RIPEMD-160** 的差異如下：

```
import hashlib                                # 匯入雜湊函數模組

msg = 'apple'.encode('utf-8') # 欲加密的訊息（需先以 utf-8 編碼）
h = hashlib.sha256(msg)        # 建立 SHA-256 物件
print('SHA-256 雜湊值:', h.hexdigest()) # 取得雜湊值（16 進制）
h = hashlib.new('RIPEMD160', msg) # 建立 'RIPEMD160' 物件
print('RIPEMD160 雜湊值:', h.hexdigest()) # 取得雜湊值（16 進制）
```



要產生雜湊值的訊息必須為經過 utf-8 編碼的字串。

SHA-256 雜湊值: 3a7bd3e2360a3d29eea436cfb7e44c735d117c42d1c1835420b6b9942dd4f1b。長度為 32 byte (因為以 16 進位呈現, 故共有 64 個字元)。

RIPEMD160 雜湊值: dfa40098c95721adb3e6b2cf23568b7f6d968694。長度為 20 byte (因為以 16 進位呈現, 故共有 40 個字元)。



RIPEMD160 雜湊函數並非屬於跨平台可用的雜湊函數, 故須以 **new()** 方法來建立。

而為什麼雜湊函數適合用來做加密呢? 我們來看看雜湊函數的幾個特性：

■ 不可逆性 (Irreversible)

雜湊函數為**單向函數 (One-way function)**，也就是說我們無法透過雜湊值，反推出原始訊息是什麼，例如我們收到一個雜湊值，我們是完全無法得知這個雜湊值代表什麼訊息。

這個概念就像是給你一個指紋，你是無法透過指紋辨識出對應的身分一樣，除非你有像警方的指紋-身分資料庫。


■ 唯一性 (uniqueness)

輸入相同的訊息一定會輸出相同的雜湊值，可以將雜湊值想像成訊息的指紋，例如我們對 'apple' 進行雜湊化，找出代表它的唯一指紋 (雜湊值)：

```
import hashlib
msg = 'apple'.encode('utf-8')      # 欲加密的訊息 (需先以 utf-8 編碼)

h1 = hashlib.sha256(msg)           # 建立 SHA-256 物件
print('h1 雜湊值:', h1.hexdigest()) # 取得 sha256 雜湊值 (16 進制)

h2 = hashlib.sha256(msg)           # 建立 SHA-256 物件
print('h2 雜湊值:', h2.hexdigest()) # 取得 sha256 雜湊值 (16 進制)
```

 輸出

```
3a7bd3e2360a3d29eea436fcfb7e44c735d117c42d1c1835420b6b9942dd4f1b
3a7bd3e2360a3d29eea436fcfb7e44c735d117c42d1c1835420b6b9942dd4f1b
```

兩次的數值都一樣，就像指紋必須具有唯一性，試想一下，如果警察在犯罪現場採集到的嫌疑犯指紋，結果對比身分後竟有 2 人有此指紋，是不是會很傷腦筋呢？

所以不同的訊息經過雜湊化後的雜湊值原則上不會重複，我們分別對 'dog' 與 'cat' 進行雜湊化試試：


```
import hashlib

msg1 = 'dog'.encode('utf-8')      # 欲加密的訊息 'dog'
h1 = hashlib.sha256(msg1)
```

[接下頁](#)

```
print('dog 雜湊值:', h1.hexdigest())    # 取得 'dog' 雜湊值

msg2 = 'cat'.encode('utf-8')           # 欲加密的訊息 'cat'
h2 = hashlib.sha256(msg2)
print('cat 雜湊值:', h2.hexdigest())    # 取得 'cat' 雜湊值
```

 輸出

```
cd6357efdd966de8c0cb2f876cc89ec74ce35f0968e11743987084bd42fb8944
77af778b51abd4a3c51c5ddd97204a9c3ae614ebccb75a606c3b6865aed6744e
```

所以在密碼學中，若訊息被偷偷竄改，則雜湊值會完全不同，這樣可以用來確保訊息的**健全性 (integrity)**。例如我們可以將「訊息」與「訊息的雜湊值」一起傳送給接收者，接收者可以將收到的訊息進行雜湊化，對比看看是否與傳送過來的雜湊值相同，就可以確認訊息是否有被竄改了：


```
import hashlib

#---- 傳送者 ----#
msg = '明天早上六點集合'    # 欲傳送的訊息
h = hashlib.sha256(msg.encode('utf-8')).hexdigest()  # 訊息雜湊值
data = {'message': msg, 'hash': h}                    # 以字典傳送訊息

#---- 中間竄改者 ----#
data['message'] = '明天早上十點集合'

#---- 接收者 ----#
recevie_msg = data['message']
verify_hash = hashlib.sha256(recevie_msg.encode('utf-8')).hexdigest()

if verify_hash == h:
    print('正確訊息: ', recevie_msg)
else:
    print('訊息已被竄改: ', recevie_msg)
```

TIPS  當然啦，傳送資料還得要加上更加嚴謹的資安處理，例如：共同金鑰、數位簽章…等等的，這些屬於密碼學的範疇，不在本討論範圍中，在這裡我們只要知道如何將雜湊函數用到我們的區塊鏈上就可以了。

■ 變異性 (Variability)

輸入訊息只要有一點點變動就會造成輸出雜湊值的巨大變化，輸入跟輸出之間完全看不出規則。我們將訊息 'apple' 改為 'Apple' 試看看：

```
import hashlib

msg1 = 'apple'.encode('utf-8')          # 欲加密的訊息 'apple'
h1 = hashlib.sha256(msg1)
print('SHA-256 雜湊值:', h1.hexdigest()) # 取得 'apple' 雜湊值

msg2 = 'Apple'.encode('utf-8')          # 欲加密的訊息 'Apple'
h2 = hashlib.sha256(msg2)
print('SHA-256 雜湊值:', h2.hexdigest()) # 取得 'Apple' 雜湊值
```

↓ 輸出

```
3a7bd3e2360a3d29eea436fcfb7e44c735d117c42d1c1835420b6b9942dd4f1b
f223faa96f22916294922b171a2696d868fd1f9129302eb41a45b2a2ea2ebbfd
```

所以無法根據訊息的內容相似性
來做暴力猜測破解。



■ 長度固定

不管輸入雜湊函數的訊息多長或多短，統一都會輸出固定長度的雜湊值（長度依據雜湊函數類型而定），例如我們輸入 'This is an apple' 與 'apple' 來試試：

```
import hashlib

msg1 = 'apple'.encode('utf-8')
h1 = hashlib.sha256(msg1)
print('apple 雜湊值:', h1.hexdigest())

msg2 = 'This is an apple'.encode('utf-8')
h2 = hashlib.sha256(msg2)
print('This is an apple 雜湊值:', h2.hexdigest())
```

[接下頁](#)



```
3a7bd3e2360a3d29eea436fcfb7e44c735d117c42d1c1835420b6b9942dd4f1b
69bb57b00a3ca55cb353e3337ef511c77e49cbb232384939eb5d71b6c631bd82
```

所以無法根據訊息的長度相似性
來做暴力猜測破解。



B-2-1 建立區塊鏈類別

要建立區塊鏈網路, 首先我們要先建立一個區塊鏈類別:

B-0.py

```
06 class Blockchain(object):  
    ...
```

繼承於 *object* 類別

TIPS 事實上所有的類別都繼承自 **object** 類別, Python 默認沒有指定繼承的類別皆自動繼承自 **object** 類別, 因此上面程式小括號中的 **object** 也可以省略。

接著我們定義初始化方法 `__init__()`:

B-0.py 續

```
06 class Blockchain(object):  
07     def __init__(self):  
08         self.current_transactions = [] # 儲存交易  
09         self.chain = []               # 儲存區塊鏈  
10         self.nodes = set()            # 用 set 儲存其他節點位址 (鄰居)  
11         self.new_block(previous_hash=1, nonce=100) # 產生創世  
                                                    (genesis) 區塊
```

`__init__(self)` 之中建立了 3 個屬性：

1 `current_transactions = []`

以串列 (List) 來儲存一筆筆的交易紀錄, 此交易紀錄是那些尚未寫到區塊上的交易紀錄, 先以 `current_transactions` 儲存起來, 等待新區塊產生後再寫入區塊, 加入區塊鏈之中。

2 `chain = []`

這就是我們的區塊鏈, 以串列 (List) 來儲存一個個的區塊, 稍後會將產生的新區塊儲存到此之中。

3 `nodes = set()`

我們用集合 (set) 來儲存參與這個區塊鏈網路的其他使用者們 (村民們), 使用 `set` 來儲存也是因為我們不希望有重複的使用者資料。

■ 產生新區塊：`new_block()`

另外在 `__init__()` 之中還執行了一個 `new_block(previous_hash=1, nonce=100)` 類別 Method, 此 Method 是用來產生一個新區塊, 而我們希望這個區塊鏈類別初始化時會先產生一個新區塊 (區塊鏈的第一個區塊也稱為**創世區塊 (Genesis Block)**), 接下來我們來定義這個 Method:

```
def new_block(self, nonce, previous_hash=None):
    ...
    ...
    return block ← 回傳新產生的區塊
```

`new_block()` 總共接收 2 個參數, 並且最後會回傳產生的新區塊, 說明如下:

1 nonce

這是一個工作量證明的數字，與稍後的挖礦演算法有關，因為我們說要產生一個新區塊必須經過挖礦的動作，而這個動作的目的就是找出某個正確的 nonce 數字，找到後即可產生新區塊，並在區塊中存入這個數字。

2 previous_hash=None

這是前一個區塊的雜湊值，預設值為 None。

完整的 new_block() 如下：

B-0.py 續

```
# -- ↓ 產生新區塊 ↓ -- #
14 def new_block(self, nonce, previous_hash=None): # 定義 Method
15     block = { 'index': len(self.chain) + 1,      # 索引為總鏈長
16               'timestamp': time(),              # 區塊產生時間
17               'transactions': self.current_transactions, # 交易紀錄
18               'nonce': nonce,                    # 工作量證明 (挖礦)
19               'previous_hash': previous_hash or self.hash(self.chain[-1])
20               # ↖ 前一個區塊的 Hash 值
21     }
22     self.current_transactions = [] # 重置當前交易清單
23     self.chain.append(block)      # 將新區塊加入鏈中
24     return block                  # 回傳新區塊
```

在此 Method 中會先建立一個字典 (dict) 作為區塊鏈的區塊 (block)，並儲存以下的資訊：

1 'index': len(self.chain) + 1

每個區塊都有一個對應的索引值，代表此區塊是屬於區塊鏈中第幾個區塊 (也與出現的順序有關)。

2 'timestamp': time()

要記錄每個區塊產生的時間點，所以要在程式碼最上面 **from time import time** 來使用時間函式。

3 'transactions': self.current_transactions

我們會將之前尚未寫到區塊中的交易記錄，儲存到新區塊之中。

4 'nonce': nonce

此即為傳入的工作量證明數字，稍後會說明礦工們怎麼透過挖礦演算法找出這個數字。

5 'previous_hash': previous_hash or self.hash(self.chain[-1])

當執行 `new_block()` 時，如果有傳入 `previous_hash`，則以傳入的數值作為前一個區塊的雜湊值；若沒有傳入 `previous_hash` (函式會用預設值 `None`)，則會執行另一個 Method：**`hash(self.chain[-1])`**，取出區塊鏈中的最後一個區塊，並將此區塊進行雜湊化取得雜湊值，稍後我們會建立這個 Method：**`hash()`**。

執行此 Method 產生新區塊後，接下來會將當前交易紀錄清空，因為紀錄已經都存進新區塊了，這樣下次新區塊再次產生時，才不會將已經儲存過的交易記錄又再次儲存到新區塊之中：

```
self.current_transactions = []      # 重置當前交易清單
```

接著就是將新區塊加入區塊鏈 (chain 串列) 之中，並回傳新區塊：

```
self.chain.append(block)           # 將新區塊加入鏈中
return block                       # 回傳新區塊
```

接下來我們來建立產生新區塊函式中用到的 Method：`hash()`

■ 產生區塊雜湊值：hash(block)

將區塊傳入此 Method 後，會回傳該區塊的雜湊值：

傳入區塊

```
def hash(block):
    ...
    return block 的雜湊值
```


而要進行雜湊化的訊息必須為經過 **utf-8 編碼的字串**，而我們的 **block** 是一個字典，所以我們得先透過 **json 套件的 dump()** 方法來將字典轉為長得很像 json 的字串後，再來進行 utf-8 編碼，可以透過以下的程式來試看看轉換的過程：

```
import json
from hashlib import sha256

block = {2: 'dog', 1: 'cat'}
block_string = json.dumps(block, sort_keys=True).encode('utf-8')
print(block_string) #(輸出) → b'{"1": "cat", "2": "dog"}'
print(sha256(block_string).hexdigest()) # 產生雜湊值
```



```
91ecf21df62be90e84490fc80621e3a1ea20c6b88b89ab606881dba7a892237f
```

了解過程後，我們在區塊鏈類別中實作這樣的 Method:hash()，而在區塊鏈類別中，產生雜湊值這個行為不需要與物件綁定，它只需要知道什麼區塊傳進來就可以了，所以我們將此 Method 定義成**靜態方法** (參照 BONUS-E 的 E-4 小節)：

B-0.py 續

```
02 import json # 匯入 json 套件
03 from hashlib import sha256 # 使用 sha256 雜湊函數
...
25 # -- ↓ 計算區塊雜湊值的方法 ↓ -- #
26 @staticmethod ← 靜態方法
27 def hash(block):
28     block_string = json.dumps(block, sort_keys=True).encode('utf-8')
29     ↖ 將 block (字典) 轉成 json 字串
30     return sha256(block_string).hexdigest()
```

到此為止我們已經完成了一個區塊鏈網路的基礎類別，而這個類別與區塊鏈網路有什麼關係呢？以及使用者要如何運行區塊鏈網路呢？

其實運行區塊鏈網路就是執行這份程式碼，建立一個區塊鏈物件，稍後會解釋這是個怎樣的觀念。不過在這之前先想想，如果每個人都執行這份程式碼，那要如何辨識出你我的分別呢？所以在這裡我們要介紹 **UUID**：

■ 村民 (節點) 的唯一身分：UUID

UUID 就是通用唯一辨識碼 (**Universally Unique Identifier, UUID**) 是一種透過標準方法產生的字串，這個字串具有唯一性，而區塊鏈中的用戶節點 (村民) 就是透過 UUID 來代表自己的身分。

若要產生一個UUID，可以使用 Python 的內建模組 `uuid`：

```
import uuid
```

此模組提供了以下 4 個 method 來產生 uuid：

1 `uuid1(node=None, clock_seq=None)`：

這是基於 Mac 位址與時間戳 (Timestamp) 來產生 UUID，由於 Mac 具有唯一性，所以不可能發生碰撞 (重複)。但 Mac 為硬體資訊，所以會有隱私性的問題。

2 `uuid3(namespace, name)`

將 namespace 與 name 結合，透過 MD5 雜湊函數產生雜湊值做為 UUID。保證了不同空間相同命名、相同空間不同命名的唯一性。但相同的命名空間與命名將產生相同的 UUID。

3 `uuid4()`：

以隨機生成器來產生 UUID。極小機率發生重複。

4 `uuid5(namespace, name)`

將 namespace 與 name 結合，透過 SHA-1 雜湊函數產生雜湊值做為 UUID。保證了不同空間相同命名、相同空間不同命名的唯一性。但相同的命名空間與命名將產生相同的 UUID。

我們來試試用 `uuid4()` 來產生 UUID：

一步一腳印

```
In [1]: from uuid import uuid4

In [2]: uid = uuid4()

In [3]: print(uid)
6906b124-462a-4f60-bf50-5852b4a5ddc9#
```

印出的 UUID 與書上不同是正常的，因為 `uuid4()` 是透過隨機生成器來產生 UUID，所以每次執行產生的 UUID 皆會不同（且不重複）。

■ 運行區塊鏈

現在我們先來建立一個區塊鏈物件，並建立自身節點的 UUID：

B-0.py 完整版

```
01 from time import time
02 import json
03 from hashlib import sha256
04 from uuid import uuid4
05
06 class Blockchain(object): # 區塊鏈類別
07     def __init__(self):
08         self.current_transactions = [] # 儲存當前交易
09         self.chain = [] # 儲存區塊鏈
10         self.nodes = set() # 用 set 儲存節點（不重複元素）
11         self.new_block(previous_hash=1, nonce=100) # 創建創世
12                                                     (genesis)區塊
13     # -- ↓ 創建新區塊 ↓ -- #
14     def new_block(self, nonce, previous_hash=None):
15         block = {'index': len(self.chain) + 1, # 索引為總鏈長
16                 'timestamp': time(), # 區塊產生時間
17                 'transactions': self.current_transactions, # 交易清單
18                 'nonce': nonce, # 工作量證明（挖礦）
```

接下頁

```

19         'previous_hash': previous_hash or self.  
                                hash(self.chain[-1])  
                                ↖ 前一個區塊的 Hash 值  
20     }  
21 }  
22 self.current_transactions = [] # 重置當前交易清單  
23 self.chain.append(block)      # 將新區塊加入鏈中  
24 return block                  # 回傳新區塊  
25  
26 # -- ↓ 計算區塊雜湊值的方法 ↓ -- #  
27 @staticmethod  
28 def hash(block):  
29     block_string = json.dumps(block, sort_keys=True).  
                                encode('utf-8')  
                                ↖ 將 block (字典) 轉成 json 字串  
30     return sha256(block_string).hexdigest() # 回傳雜湊值  
31  
32  
33 # -- ↓ 運行區塊鏈網路 ↓ -- #  
34 node_uuid = str(uuid4()).replace('-', '') # 為此節點產生一個 UUID  
35 blockchain = Blockchain()                 # 產生 Blockchain 物件  
36 print('此節點的 UUID:', node_uuid)  
37 print('當前交易:', blockchain.current_transactions)  
38 print('區塊鏈:', blockchain.chain)  
39 print('區塊鏈網路用戶:', blockchain.nodes)

```

程式最後輸出了：

- **此節點的 UUID: 70a9843d0ee446ad9191bfc1bef8e689**

此節點的 UUID, 我們去除了 uuid4() 產生的 '-', 讓它看起來是單純的英數字串 (每個人都不一樣)。

- **當前交易: []**

因為我們尚未建立任何交易, 所以為空串列。

- **區塊鏈: [{'index': 1, 'timestamp': 1546678801.6103268, 'transactions': [], 'nonce': 100, 'previous_hash': 1}]**

這是物件建立時, 初始化產生的創世區塊, 存於區塊鏈之中。

● 區塊鏈網路用戶: set()

尚未加入其他用戶節點的位址, 目前為空。

這樣我們就運行了一個基本的區塊鏈網路了, 有節點的 UUID、一個創世區塊、但其他的交易清單、網路用戶都還尚未有資料進來, 因為我們還未建立這些功能的 Method, 所以接下來我們就在類別中繼續加入一些關於區塊鏈的功能。

■ 建立交易方法: new_transaction()

我們說在區塊鏈網路中, 村民們可以建立交易、也可以透過挖礦來賺取虛擬貨幣, 這些行為我們可以在類別中建立以下的 Method 來進行:

```
def new_transaction(self, sender, recipient, amount):
    ...
    return 區塊鏈中的最後一個區塊位置 + 1
```

此 Method 有 3 個參數, 說明如下:

參數	說明
sender	匯款人
recipient	收款人
amount	匯款金額

而發起交易後, 會回傳區塊鏈中, 最後一個區塊的索引|位置 + 1, 也就是這筆交易紀錄目前尚未被寫入區塊中, 而它必須要等到新區塊被產生後, 才能寫入區塊中, 加到區塊鏈內, 而這個新區塊的索引值就是整個區塊鏈最尾端位置再 +1。

我們來看看這個 Method 的內容:

已放入 B-1.py 之中

```
31     # -- ↓ 交易方法 ↓ -- #
32     def new_transaction(self, sender, recipient, amount):
33         # -- ↓ 建立交易 ↓ -- #
34         trans = {'sender': sender,
35                 'recipient': recipient,
```

[接下頁](#)

```

36         'amount': amount}
37     self.current_transactions.append(trans) ← 將交易加入當前
                                              交易紀錄清單中
38     # -- ↓ 回傳將被新增到的區塊 (下一個待挖掘的區塊) 的索引 ↓ -- #
39     return self.last_block['index'] + 1

```

我們建立一個字典 `trans` 來作為一筆交易，其儲存了匯款人、收款人、匯款金額；並將此交易添加 (`append`) 到當前交易紀錄清單中 (`current_transactions`)，等待新區塊被挖出來後，可以寫到區塊之中。

而此 Method 最後透過以 `@property` 定義的外部屬性 `last_block` 來取得區塊鏈最尾端的區塊 (是一個字典)，再以鍵 `'index'` 來取得索引位置，我們來看看這個外部屬性：

■ 取得最後一個區塊：`last_block()`

在 BONUS-E 的 E-6 節有提到，在類別中可以建立一個方法，並在此方法上面標註 `@property` 作為外部屬性的取用方式：

B-1.py 續

```

41 @property
42 def last_block(self):
43     return self.chain[-1] ← 回傳鏈的最後一個區塊

```

此方法很簡單，主要就是要回傳鏈中最後一個區塊，而標註了 `@property` 讓我們可以用以下的方式來使用，讀者可以在 B-1.py 執行看看這個功能：

last_block 之後不用加 ()

```
物件.last_block['index']
```

有了交易方法後，我們就可以透過此方法來讓村民們發起交易、或是透過此方法來發送虛擬貨幣獎勵給礦工，接下來我們就來看看，到底礦工們如何在此區塊鏈網路中進行挖礦。

B-2-2 挖礦演算法：產生新區塊、獲取虛擬貨幣獎勵

前面我們已經在類別中加入了產生新區塊、發起交易的 Method 了，現在來看看礦工們到底到滿足什麼條件，才能產生新區塊。

參與區塊鏈網路的礦工們進行**挖礦**，其實是要執行區塊鏈類別中的一個**挖礦 Method**，此 Method 的用意就是要證明這個礦工的工作量，但我們要先了解礦工們到底要做些什麼工作呢？

其實礦工們要做的事就是用暴力演算來找出一個數字 nonce，每一個區塊都有各自的一個 nonce，礦工們要產生新區塊，就是得找到新區塊專屬的那個 nonce，那…怎麼找呢？這就是挖礦 Method 的工作了，我們來看看這個 Method：`find_nonce()`

■ 挖礦：`find_nonce()`

```
def find_nonce(self, last_nonce):
    ...
    return nonce    # 回傳找到的nonce（新區塊的 nonce）
```

↑ 上一個區塊的 nonce

此 Method 會用到上一個區塊的 **nonce** 數字，也就是新區塊的 nonce 會與上一個區塊的 nonce 有關聯性，所以這樣區塊與區塊才會前後相接，我們來看看此 Method 的內容：

已放入 B-2.py 之中

```
46 # -- ↓ 挖礦演算法：工作量的證明，尋找 nonce ↓ -- #
47 def find_nonce(self, last_nonce):
48     nonce = 0 # 從 0 開始找
49     while self.valid_nonce(last_nonce, nonce) is False:
50         nonce += 1
51     return nonce
```

此 Method 將 nonce 從 0 開始, 接著進入 while 迴圈中, 在此迴圈中會執行另一個 **Method: valid_nonce()** 來驗證這個 nonce 是否為正確的 nonce, 若不是, 則 nonce +1 遞增, 然後繼續驗證, 直到找到正確的 nonce, 我們來看看 valid_nonce() 是如何驗證 nonce 的:

■ 驗證 nonce : valid_nonce()

此方法一樣是個靜態方法, 有 2 個參數: 上一個區塊的 nonce、與正要驗證的 nonce:

```
@staticmethod
    上一個區塊的 nonce      嘗試的 nonce
    valid_nonce(last_nonce, nonce):
        ...
        return 是否正確
```

而要如何驗證呢? 其實方法可以自訂 (難易度), 例如我們規定可以將上一個區塊的 nonce 與新區塊的 nonce 前後串接組成新字串, 然後將它進行雜湊化取得一個雜湊值, 而這個雜湊值的開頭前 4 個數字都要是 0, 我們來看看這個 Method 的內容:

已放入 B-2.py 之中

```
54 @staticmethod
55 def valid_nonce(last_nonce, nonce):
56     guess = f'{last_nonce}{nonce}'.encode()
57     guess_hash = sha256(guess).hexdigest()
58     return guess_hash[:4] == '0000' ➡ 輸出 若找到則回傳 True
```

我們可以將上述的內容結合, 來實際挖挖看礦, 找到創世區塊 (nonce=100) 的下一個區塊的 nonce 是多少:

■ 用自訂函式: mine() 進行挖礦

在此建立一個可以讓村民進行挖礦的自訂函式 **mine()**, 執行此自訂函式後, 會開始尋找創世區塊 (nonce=100) 的後一個新區塊的 nonce 是多少:

B-2.py

```

01 from time import time
02 import json
03 from hashlib import sha256
04
05 class Blockchain(object):
06     def __init__(self):
07         self.current_transactions = [] # 儲存當前交易
08         self.chain = []                # 儲存區塊鏈
09         self.nodes = set()             # 用 set 儲存其他節點
10         self.new_block(previous_hash=1, nonce=100) # 創建創世(genesis)區塊

```

…同 B-1.py

```

46     # -- ↓ 挖礦演算法：工作量的證明，找到 nonce ↓ -- #
47     def find_nonce(self, last_nonce):
48         nonce = 0 # 從 0 開始
49         while self.valid_nonce(last_nonce, nonce) is False:
50             nonce += 1
51         return nonce
52
53     # -- ↓ nonce 的驗證 ↓ -- #
54     @staticmethod
55     def valid_nonce(last_nonce, nonce):
56         guess = f'{last_nonce}{nonce}'.encode() # 前後區塊 nonce 相串接
57         guess_hash = sha256(guess).hexdigest()
58         return guess_hash[:4] == '0000'
59
60 #---- ↓ 運作區塊鏈 ↓ ----#
61 blockchain = Blockchain() # 產生 Blockchain 物件
62 print('當前交易:', blockchain.current_transactions)
63 print('區塊鏈:', blockchain.chain)
64 print('區塊鏈網路用戶:', blockchain.nodes)
65 print('區塊鏈最後一個區塊索引位置:', blockchain.last_block['index'])
66 #---- ↓ 挖礦自訂函式 ↓ ----#
67 def mine():
68     last_nonce = blockchain.last_block['nonce'] # 取得最後一個區塊的 nonce
69     new_nonce = blockchain.find_nonce(last_nonce) # 執行挖礦 Method
70     print('新區塊的 nonce:', new_nonce)
71
72 mine() # 執行挖礦

```



新區塊的 nonce: 35293

可以看到新區塊的 nonce 被找出來了，而我們也可以讓這個 nonce 變得難找一點，例如我們要求要找到前五個數字都為 0：`guess_hash[:5] == '00000'`。運行後可以看到找到的 nonce 為 888273，運行較長的時間才找到，找到的 nonce 數字也大很多。

一般區塊鏈都會設計成區塊會越來越難挖喔，以免區塊產生速度太快。



礦工找到 nonce 後，接下來就有資格執行 Method: new_transaction() 來發起匯款虛擬貨幣給自己的一筆交易，並且執行可以產生新區塊的 Method: new_block()。我們繼續在 mine() 之中加入這兩件事：

已放入 B-2.py 之中

```

68 #---- ↓ 挖礦自訂函式 ↓ ----#
69 def mine():
70     last_nonce = blockchain.last_block['nonce']
71     new_nonce = blockchain.find_nonce(last_nonce)
72     print('產生新區塊的 nonce:', new_nonce) ← 算出新區塊的 nonce 程式
                                                才 73 會繼續往下執行
74     # -- ↓ 給予獎勵:發起一筆給自己的交易 ↓ -- #
75     blockchain.new_transaction(sender='0', # 匯款人為 0 代表
                                    # 是新挖出來的幣
36                                     recipient=node_uuid, # 收款人為自己的 UUID
37                                     amount=1, # 一個幣
76     )
77
78     # -- ↓ 產生新區塊 ↓ -- #
79     newBlock = blockchain.new_block(new_nonce) # 執行產生新區塊的 Method
80     response = { ← 用字典做一個新區塊的資訊, 稍後可以回傳給村民看
81         'message': "New Block Forged",
82         'index': newBlock['index'],
83         'transactions': newBlock['transactions'],
84         'nonce': newBlock['nonce'],
85         'previous_hash': newBlock['previous_hash'],
86     }
87

```

接下頁

```

88     return response
89
90 response = mine() # 執行挖礦
91 print('新區塊資訊：', response)

```



```

新區塊資訊： {'message': 'New Block Forged',
'index': 2, 'transactions': [{'sender': '0', 'recipient': '自己的
UUID', 'amount': 1}], 'nonce': 35293, 'previous_hash': '8213505e4da
c8da713e18b3bf56e6e78f5d49a3f5a12e0a9b64d08962094759c'}

```

執行後可以看到新區塊已經被挖出來了，並且印出新區塊的資訊：包含了新區塊的索引位置、寫入的交易紀錄（目前只有匯給礦工（自己）的獎勵）、新區塊的 nonce、前一個區塊的雜湊值。

B-3 用 Flask 架設區塊鏈網站伺服器

前面我們建立了一個區塊鏈類別，並透過建立區塊鏈物件來進行挖礦、交易、產生新區塊…等等的 Method 來運行區塊鏈的基本功能，但這樣僅能在本地端執行（自嗨），我們還必須與其它節點（用戶）共同運行這個機制才有意義。

所以接下來我們要將此程式碼以 **Flask Web App** 的網站形式來建立**區塊鏈網站伺服器**，如此節點（用戶）便可透過 **HTTP 請求**來對這個網站伺服器進行區塊鏈的功能請求。



稍後我們也會透過 Postman 這個軟體來對網站進行 HTTP 請求的測試。

B-3-0 Flask 簡介

Flask 是一個用來開發輕量級網站的 Python 微框架 (micro-framework)，這表示透過 Flask 來建立網站是非常容易的事，而且此框架不會先替你太多事，所以你可自行擴充想要的功能，使用的彈性很大。



上圖為 Flask 官方的 Logo，可以看到右下方有一段文字：「web development one drop at a time」。意味著用 Flask 來建構網站，就像是一瓶水，一次只倒出一滴的概念，每一滴都是依據開發者的需求而倒出，一滴滴的建構出客製化的網站。

B-3-1 建立 Flask 網站

要透過 Flask 來建立網站非常簡單快速，通常 Anaconda 已經幫我們安裝好 Flask 了，所以我們不需要再額外安裝，現在馬上來試試建立一個簡單的網站：

如果出現 `...\.click\utils.py... UnsupportedOperation: not writable` 的錯誤，請開啟 Anaconda prompt 視窗執行 `pip install --upgrade click` 來更新 click 套件至最新版本，然後再重新啟動 Spyder。



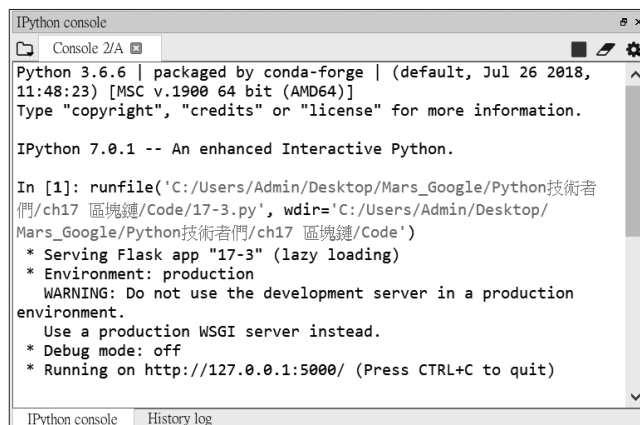
B-3.py

```
01 from flask import Flask # 匯入 Flask 類別
02 app = Flask(__name__) # 透過 Flask 類別建立一個物件
03
04 @app.route('/') # 建立路由：網站為使用者指路(路徑)
05 def hello():
06     return 'Hello Flask! '
07
08 if __name__ == '__main__':
09     app.run() # 啟動網站
10     # app.run(debug=True) # 以 debug 模式啟動網站
11     print('網站已結束')
```

程式說明

- 02 使用 Flask 類別來建立一個 Flask 物件，以 `__name__` 做為引數傳入，這是 Python 中的特殊變數，當這份程式碼被直接執行時，`__name__ == '__main__'`；而若是這份程式碼被當成模組，被其他程式碼匯入使用時，`__name__` 會變成模組名稱。而透過 `__name__` 來建立 Flask 物件已是約定成俗的用法。
- 04 Flask 透過 `@app.route()` 裝飾器來定義這些網站有哪些路由可以使用，用戶端可以根據路由指示的路徑，進行 HTTP 的請求。例如在 `@app.route()` 之中放入了 `'/'`：代表這個網站的根路徑（首頁），用戶端在瀏覽器中輸入根路徑後（發出請求），網站就會執行 `@app.route('/')` 下方的自訂函式：`hello()`。而我們也可以定義其他的路徑，例如：`/new`。
- 05 當被導到 `hello()` 函式後，函式直接執行了 `return 'Hello Flask!'` 回傳給用戶端的瀏覽器去顯示。
- 08 此行代表這份程式碼需要被直接執行（不可作為其他程式碼的模組來執行），才會運行 09~11 行的程式區塊。
- 09 執行 `run()` 方法就會啟動這個網站。
- 10 在 `run()` 方法中可以加入 `debug=True`，讓網站以 debug 模式啟動，方便除錯以及網站程式碼有變動時，網站會自動重啟，但是當網站要正式營運時，千萬要把 debug 模式關閉，以免洩漏網站內部資訊。
- 11 執行 `run()` 啟動網站後，會進入循環，所以當網站結束執行後，11 行才會執行。

執行後，在 IPython 可以看到如右的畫面：



```

IPython console
Console 2/A
Python 3.6.6 | packaged by conda-forge | (default, Jul 26 2018, 11:48:23) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.0.1 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/Admin/Desktop/Mars_Google/Python技術者們/ch17 區塊鏈/Code/17-3.py', wdir='C:/Users/Admin/Desktop/Mars_Google/Python技術者們/ch17 區塊鏈/Code')
* Serving Flask app "17-3" (lazy loading)
* Environment: production
WARNING: Do not use the development server in a production environment.
Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

IPython console  History log

```

這個畫面的最後一行告訴了我們：網站正以 `http://127.0.0.1:5000` (預設) 的位址被啟動，用戶端可以透過此路徑發出請求、若想關閉網站，請在 IPython 中按下『`Ctrl + C`』。

我們現在打開瀏覽器，試著對此網站發出請求：



執行後也可以在 IPython 看到網站接收到的請求資訊：

```
127.0.0.1 -- [06/Jan/2019 15:44:28] "GET / HTTP/1.1" 200 -
```

Annotations for the log line:

- 請求時間 (Request Time): points to the timestamp `[06/Jan/2019 15:44:28]`.
- 請求類型：GET 與狀態碼 200 (表示 OK) (Request Type: GET and Status Code 200 (表示 OK)): points to `"GET / HTTP/1.1" 200`.
- 請求來源 (本地端) (Request Source (Local)): points to the IP address `127.0.0.1`.

而我們也看到了 `run()` 以本地位址 `127.0.0.1` 做為網站位址做為測試使用 (只監聽本地端的請求)，若要讓外部用戶端來使用此網站 (網站正式上線)，可以在 `run()` 設定監聽位址：

```
app.run(host='0.0.0.0', port=5001)
```

這樣代表以通訊埠 5001 來監聽所有位址。

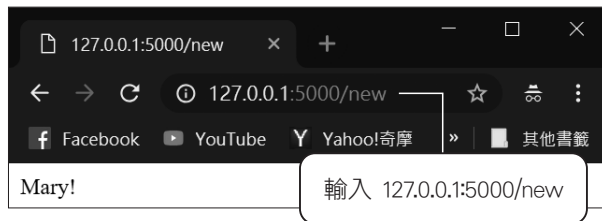
■ 新增路由 (route)

而我們當然還可以替這個網站添加其他路由，讓用戶端可以根據路由發出不同路徑請求，來使用各種網站的功能，例如我們新增一個路由 `/new`，並且可以指定用戶端需使用 **HTTP GET** 方法來進行請求：

B-4.py

```
01 from flask import Flask
02 app = Flask(__name__)
03
04 @app.route('/')
05 def hello():
06     return 'Hello Flask!'
07
08 @app.route('/new', methods=['GET'])    # 新路由
09 def name():
10     return 'Mary!'
11
12 if __name__ == '__main__':
13     app.run()
```

執行後，並在瀏覽器
中輸入新的請求路徑 /new
看看：



稍後我們就會將區塊鏈的功能，透過建立各種路由來讓節點進行路徑請求，使用功能。

■ 對網站發出 HTTP POST 請求

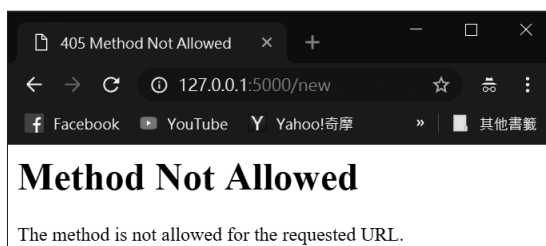
我們當然也可以在裝飾器中去定義這個路徑需要透過 **HTTP POST** 請求才可以（因應用戶端可以傳送資料給網站的需求，例如稍後我們可以將別的節點位址傳給網站），我們來試試將 B-4.py 程式碼中的 /new 路徑改為規定以 POST 請求：

已放入 B-5.py 之中

```
08 @app.route('/new', methods=['POST'])
09 def name():
10     return 'Mary!'
```

需使用 HTTP POST 進行請求

然而執行 B-5.py 運行網站後，以瀏覽器發出 /new 路徑請求後會看到如右的結果：



這是因為瀏覽器是以 HTTP GET 對這個路徑發出請求，而這個請求方法不被網站允許 (需以 HTTP POST)；再看看 IPython 顯示的請求資訊：

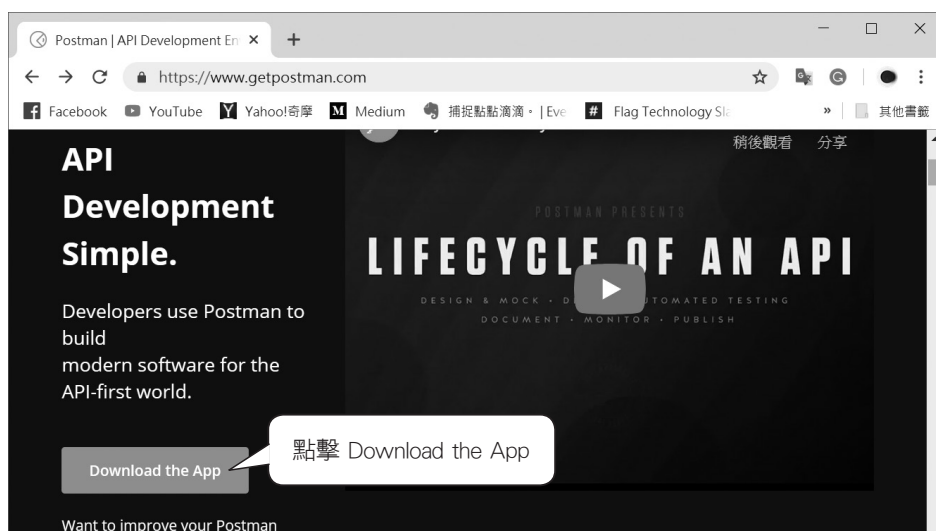
```
127.0.0.1 - - [06/Jan/2019 16:25:35] "GET /new HTTP/1.1" 405
```

可以看到最尾端網站回應了狀態碼 405：代表請求的方法 (GET) 不能被用於相對應的路徑 (/new)。

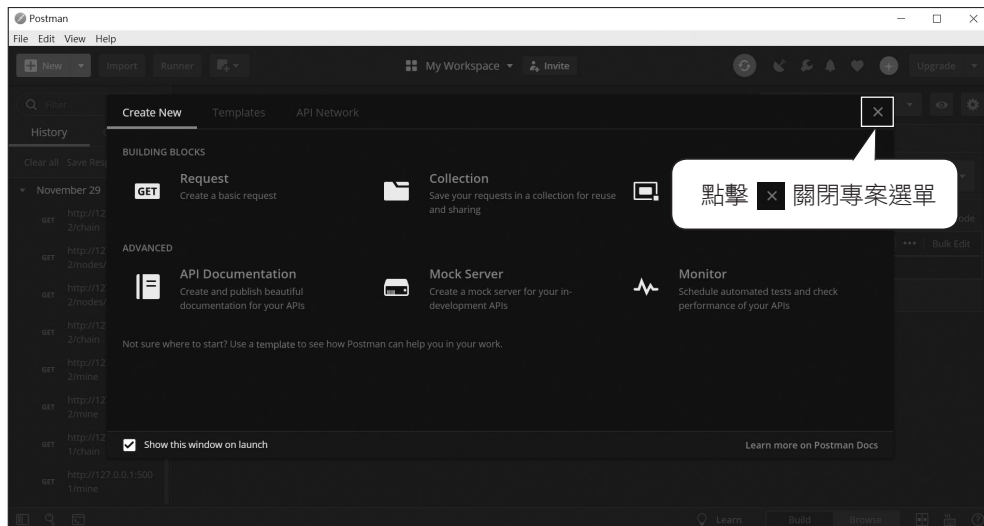
所以稍後我們將介紹 **Postman** 這套用於測試網站請求的軟體工具，方便於我們發出各種類型的請求方法。

B-3-2 Postman

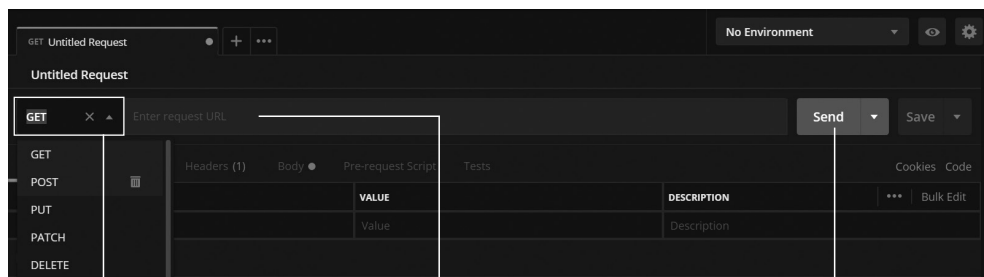
接下來我們來試試透過 Postman 來對我們的 Flask 網站發出請求，請先至 Postman 官方網站下載軟體：<https://www.getpostman.com/>：



接著依照電腦作業系統來下載軟體，下載完後點擊即可自動安裝並啟動程式。Postman 軟體介面如下，(請先將軟體一打開時會跳出來的登入帳號及專案選單介面先關閉：



關閉後可以看到中間的部分如下，我們可以透過此區域的輸入框及選單來發送請求：

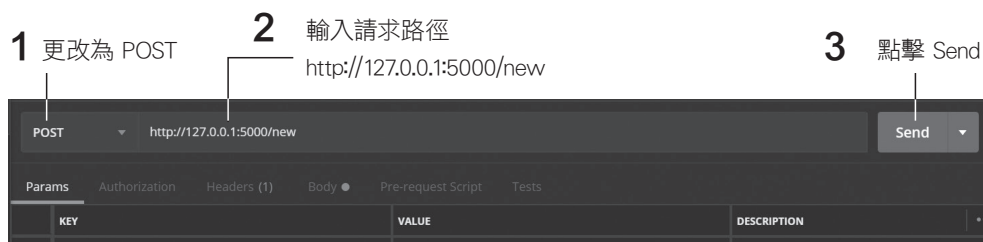


可以選擇請求方法

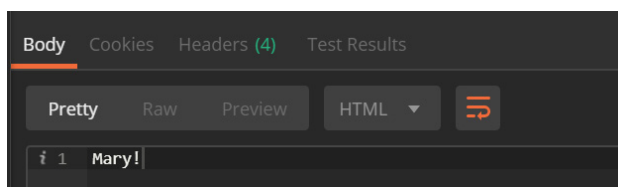
輸入網站請求路徑

點擊 Send 即發送請求

我們再次運行剛剛有 POST 路由的 Flask 網站 (B-5.py)，並透過 Postman 來發送 HTTP POST 請求：



發出請求後，即可在 Postman 下方的 Body 區域看到網站回傳的 Mary!:



現在我們可以對網站發出 POST 請求了，回想一下，在本書第 16 章：人臉辨識章節中，我們也使用 requests 模組對 Azure 網站伺服器發出 POST 請求，將一些要建立人臉辨識所需的資訊，傳給 Azure 網站伺服器。

所以，概念一樣，現在請將我們用 Flask 建立的網站，想像成 Azure 網站伺服器，我們也可以在 Flask 網站中建立一個可以接收用戶端資料的路徑。

例如我們建立一個路由，可以讓區塊鏈用戶端（節點）將別人的節點位址傳過來，而要在網站中接收用戶端 POST 過來的資料，需用到 flask 套件中的 request 方法。

與 Python 的 requests 套件不同喔！沒有 s



我們規定用戶端必須將節點位址以

```
{
  "nodes":["http://127.0.0.1:5002"]
}
```

的形式傳過來，回想一下，我們在對 Azure 伺服器發送 POST 請求傳遞資料時，是不是也都要將資料以它規定的形式包起來，以請求主體進行發送呢？概念都是一樣的！

匯入此方法可以將非字串的
資料轉成 json 格式的字串

```
from flask import Flask, request, jsonify
```

要額外匯入此方法

接著我們就可以在自訂函式中撰寫接收資料的程式碼：

已放於 B-6.py 之中

```
12 # -- ↓ 路由：註冊節點 ↓ -- #
13 @app.route('/nodes/register', methods=['POST'])
14 def register_nodes():
15     values = request.get_json() # 接收資料，values 為字典型別
16     nodes = values['nodes']    # 取出字典中鍵為 'nodes' 的資料（串列）
17
18     if nodes is None: # 如果找不到資料，則代表用戶端傳遞資料的格式錯誤
19         return '請輸入正確的節點位址', 400
20     return jsonify(values) # 將字典（非字串資料）轉成 json 字串後傳回
```

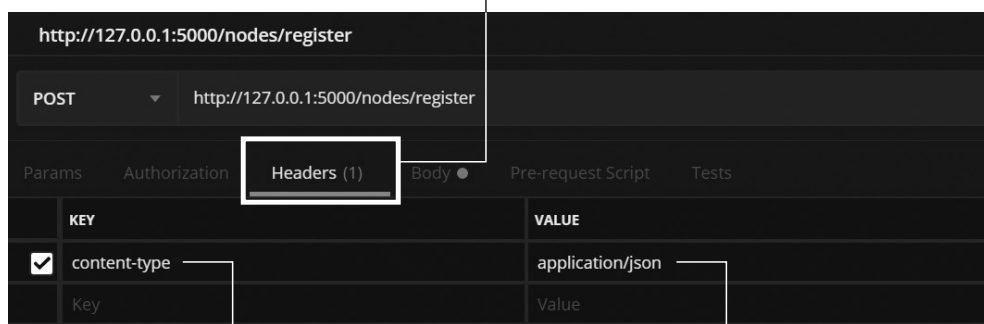
請執行 B-6.py 啟動網站後，到 Postman 發送上面的節點位址給網站看看，還記得在第 16 章人臉辨識章節時，要發送帶有請求主體的 HTTP POST 時，是不是都還要額外在請求標頭中設定一下請求主體的內容類型 ('Content-Type': 'application/json') 呢，看下圖回憶一下吧：

```
key = '3aebb4c9bef249aba9dad2d3' ← 你的 key
headers_json = {'Ocp-Apim-Subscription-Key': key,
                'Content-Type': 'application/json'} ← body 內容類型，使用 MIME 格式（可參考第 13 章）
response = requests.put(請求路徑, headers= headers_json, 請求主體)
```

以 headers 來指定請求標頭參數

我們也要在 Postman 設定請求主體的內容類型，一樣為 'application/json'，請在 Postman 中選擇 Header 分頁，這裡就是用來設定請求標頭的地方，請依下圖指示，在 KEY 欄位中輸入：content-type、在 VALUE 欄位中輸入：application/json：

1 切換到 Headers



2 輸入 content-type

3 輸入 application/json

接下來我們要發送的資料會放在請求主體中，還記得我們在第 16 章把請求主體放在一個字典中嗎？看下圖回憶一下吧：

鍵的名稱 name、userData 是 Azure API 規定的固定識別字

```
body = {'name': '旗標科技公司', 'userData': '位於台北市'}
body = str(body).encode('utf-8') ← 將字典轉為 utf-8 編碼的字串
```

```
response = requests.put(請求路徑, 請求標頭, data=body)
```

以 data 來指定請求主體參數

我們現在一樣要在 Postman 中設定請求主體 (body) 是什麼，請在 Postman 中切換到 Body 分頁，並在分頁中輸入規定格式的節點位址資料：

1 切換到 Body

2 選擇 raw



3 在下方輸入區輸入節點位址

接著在請求路徑欄位輸入 Flask 網站接收節點資料的路徑：

1 選擇 POST

2 輸入 http://127.0.0.1:5000/nodes/register

3 點擊 Send

4 在 Params/Body 可以看到網站成功接收，並回傳我們剛剛輸入的節點位址

待會我們會替網站中接收節點位址的自訂函式 `register_nodes()` 添加更多程式碼，現在我們只要知道網站已經可以接收其他人的節點位址就可以了。

■ 建立挖礦路由

現在我們來將 **B-2.py** 區塊鏈程式碼改以 Flask 網站的形式發佈，類別的內容皆不需要更動，只有執行挖礦改由 Flask 的**路由** `/mine` 來觸發，下面程式碼加粗的部分就是變更的地方：

```

B-7.py
01 from time import time
02 import json
03 from hashlib import sha256
04 from uuid import uuid4
05 from flask import Flask, request, jsonify    # 匯入 flask 套件
06
07 class Blockchain(object):

    ...區塊鏈類別內容（與 B-2.py 相同）
    
```

接下頁

```

62 #---- ↓ 運作區塊鏈 ↓ ----#

63 app = Flask(__name__) # 產生一個 Flask 物件
64 node_uuid = str(uuid4()).replace('-', '') # 為此節點產生一個 UUID
65 blockchain = Blockchain() # 產生 Blockchain 物件
66 print('此節點的 UUID:', node_uuid)
67 print('當前交易:', blockchain.current_transactions)
68 print('區塊鏈:', blockchain.chain)
69 print('區塊鏈網路用戶:', blockchain.nodes)
70 print('區塊鏈最後一個區塊索引位置:', blockchain.last_block['index'])
71
72 # -- ↓ 路由：挖礦 ↓ -- #
73 @app.route('/mine', methods=['GET']) # 在原挖礦自訂函式的上方加上路由裝飾器
74 #---- ↓ 挖礦自訂函式 ↓ ----# # 路徑為 /mine, 方法為 GET
75 def mine():
76     last_nonce = blockchain.last_block['nonce']
77     new_nonce = blockchain.find_nonce(last_nonce)
78     print('產生新區塊的 nonce:', new_nonce) # 算出新區塊的 nonce 程式
                                              # 才會繼續往下執行
79     # -- ↓ 給予獎勵：發起一筆給自己的交易 ↓ -- #
80     blockchain.new_transaction(sender='0', # 匯款人為 0 代表是新挖
81                                 recipient=node_uuid, # 收款人是自己的 UUID
82                                 amount=1, # 一個幣
83     )
84     # -- ↓ 產生新區塊 ↓ -- #
85     newBlock = blockchain.new_block(new_nonce) # 執行產生新區塊的 Method
86     response = {
87         'message': "New Block Forged",
88         'index': newBlock['index'],
89         'transactions': newBlock['transactions'],
90         'nonce': newBlock['nonce'],
91         'previous_hash': newBlock['previous_hash'],
92     }
93     return jsonify(response), 200 # 將挖出來的區塊資訊回傳給用戶端查看
94                                ↗ 也可以再多回傳 HTTP 狀態碼給用戶端
95 if __name__ == '__main__':
96     app.run() # 啟動網站

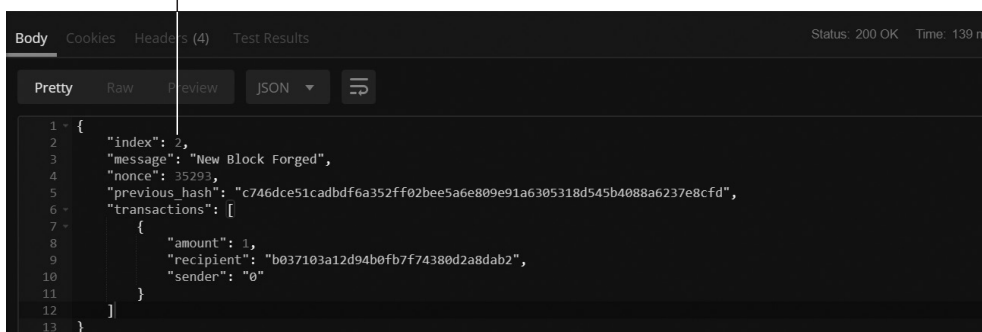
```

一樣執行後，我們到 Postman 去進行 **/mine** 的路徑請求：



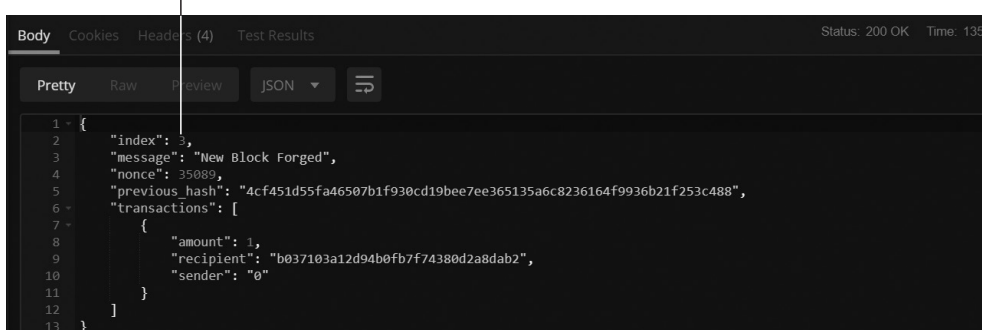
發出請求後，就會執行我們在路由裝飾器下方的挖礦自訂函式 `mine()`，這個自訂函式跟先前的幾乎相同，唯一不同的是最後回傳新區塊的資料，改由 Flask 的 `jsonify()` 方法來回傳。挖完礦後，即可在 Postman 下方看到網站回傳的新區塊資料：

Index 為 2，第 2 個區塊被挖出來了



我們還可以繼續發出 `/mine` 路徑請求，繼續挖礦，如下圖可以看到，再次發出請求後，回傳的新區塊索引為 3，代表這是區塊鏈的第 3 個區塊：

Index 為 3，第 3 個區塊被挖出來了



■ 建立查看區塊鏈資訊的自訂函式與路由

上述內容執行了 2 次挖礦，所以現在區塊鏈中有 3 個區塊了（含創世區塊），現在我們來替這個區塊鏈網站新增一個查看區塊鏈中的區塊資訊的自訂函式 `check_chain()` 以及替它建立路由 `/chain`：

B-8.py

```
88 # -- ↓ 路由：查看區塊鏈中的區塊資訊 ↓ -- #
89 @app.route('/chain', methods=['GET'])
90 def check_chain():
91     response = {
92         'chain': blockchain.chain, # 區塊鏈
93         'length': len(blockchain.chain), # 區塊鏈長度
94         'last_bk_t': blockchain.last_block['timestamp'],
95     }
96     return jsonify(response), 200
```

取得鏈的最後一個 block 的 timestamp

也可以再多回傳 HTTP 狀態碼給用戶端

執行 B-8.py 啟動網站後，我們到 Postman 發送 `/chain` 請求，看看結果：

TIPS

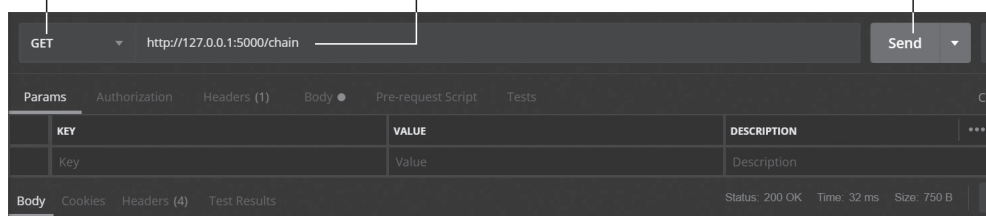


記得重複 B-7.py 的動作，先連到 `/mine` 路徑挖幾個新區塊到區塊鏈中再進行查看。

1 選擇 GET

2 輸入 `http://127.0.0.1:5000/chain`

3 點擊 Send



執行後，我們就可以在 Postman 下面看到網站回傳的整個區塊鏈的資訊：



B-4 共識演算法

現在我們來思考一個問題，若每個節點各自運行了這份程式碼，各自進行挖礦，這樣會出現一個問題：大家的區塊鏈都不一樣，這違背了我們當初說的概念：每個村民的帳本（區塊鏈）都相同這件事。

為了解決不同節點可能會擁有不同鏈的問題，我們要在區塊鏈類別中建立一個共識演算法，規定區塊鏈網路中有效最長鏈才是實際的鏈：

- **有效**：區塊與區塊之間的 nonce 皆為正確內容（鏈中沒有區塊被惡意替換）。稍後會建立一個 Method: valid_chain() 來進行區塊鏈的檢驗。
- **最長**：區塊鏈中的區塊數量最多，也就是 len(chain) 長度最大才是最正確的。

共識演算法的核心概念就是：我這個節點會與其它節點進行區塊鏈的比較（看誰比較長，若一樣長的話，看誰最後一個區塊比較早被產生）。

■ 加入其他節點的位址：add_node()

若要做到這件事，第一步就是我知道其它節點（村民）是誰（其他的節點伺服器位址），這樣我才可以向它們索取它們的區塊鏈內容（發出查看他人的區塊鏈內容的請求 /chain）來與自己的區塊鏈進行比較。

現在我們就在區塊鏈類別中，建立一個可以加入其它節點位址的 Method：
add_node()

B-9.py 此 Method 放置於區塊鏈類別中

```
62 # -- ↓ 加入其他節點位址的 Method ↓ -- #
63 def add_node(self, address):
64     self.nodes.add(address) # 將其他節點的位址加入到區塊鏈 node 清單中
```

這個 address 從哪來呢？就是我們在 B-6.py 中，練習接收用戶端透過 POST 發送過來的資料（其他節點的位址）例如 127.0.0.1:5001、127.0.0.1:5002…：

位於 B-6.py 中

```
12 # -- ↓ 路由：註冊節點 ↓ -- #
13 @app.route('/nodes/register', methods=['POST'])
14 def register_nodes():
15     values = request.get_json() # 接收資料，values 為字典型別
16     nodes = values['nodes']     # 取出字典中鍵為 'nodes' 的資料
17     if nodes is None:           # 如果找不到資料，則代表用戶端傳遞資料的格式錯誤
18         return '請輸入正確的節點位址', 400
19     return jsonify(values)      # 回傳非字串資料
```

其中 nodes 就是傳過來的節點位址，而我們現在要更進一步，收到正確的節點位址後，將位址加入到區塊鏈的節點清單 (blockchain.nodes) 中：

位於 B-9.py 之中

```
109 # -- ↓ 路由：加入其他節點位址 ↓ -- #
110 @app.route('/nodes/register', methods=['POST'])
111 def register_nodes():
112     values = request.get_json() # 接收資料，values 為字典型別
113     nodes = values['nodes']     # 取出字典中鍵為 'nodes' 的資料
114     if nodes is None:           # 如果找不到資料，則代表用戶端傳遞資料的格式錯誤
115         return '請輸入正確的節點位址', 400
116     # -- ↓ 提供節點正確，將所有提供的節點註冊到區塊鏈的節點清單中 ↓ -- #
117     for node in nodes:
118         blockchain.add_node(node) # 加到區塊鏈的節點清單中
119     response = {                 # 用來回傳給用戶端的資訊
```

[接下頁](#)

```

120         'message': 'New nodes have been added',
121         'total_nodes': list(blockchain.nodes),
122     }
123     return jsonify(response)      # 回傳非字串資料

```

現在我們可以執行 B-9.py 啟動網站，並使用 Postman 對 `/nodes/register` 路徑發出 HTTP POST 請求，新增其他節點的位址到區塊鏈節點清單中：

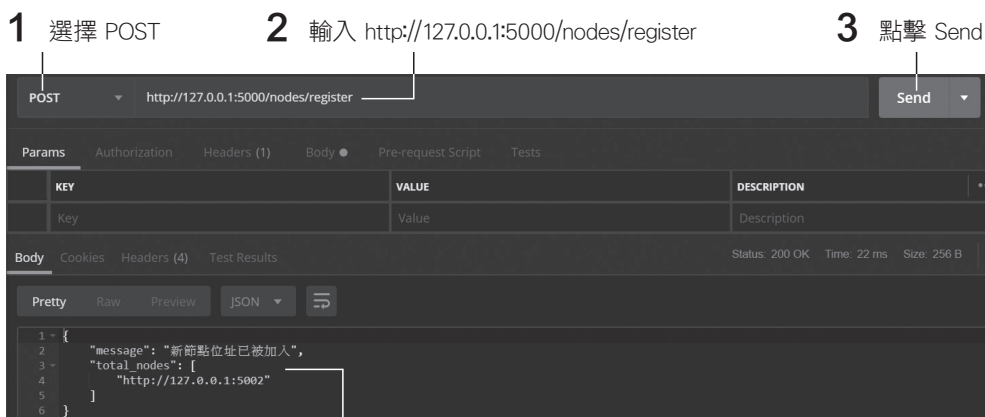


假設我們得知我們的鄰居節點的位址為：`"http://127.0.0.1:5002"`



2 在下方輸入區鍵入節點位址

接著就可以發送 POST 請求，將其他節點的位址加入區塊鏈節點清單中：



4 可以看到新節點位址已被加入

■ 共識演算法：resolve_conflicts()

現在假設區塊鏈網路中有兩個節點：`http://127.0.0.1:5000` 與 `http://127.0.0.1:5002`。而這兩個節點已經都將對方加入到自己的區塊鏈節點清單中了，接下來我們各自要處理與對方的共識問題，也就是比較誰的鏈最長、最正確。請在區塊鏈類別中新增處理共識性的 Method：`resolve_conflicts()`：

此 Method 新增於 B-10.py 之中

```

66 # -- ↓ 共識演算法：找尋區塊鏈網路中的最長鏈 ↓ -- #
67 def resolve_conflicts(self):
68     neighbours = self.nodes          # 取得所有的鄰居節點
69     new_chain = None                  # 用來記錄是否有找到新鏈
70     max_length = len(self.chain)     # 先記錄自己的鏈的長度
71     my_last_bk_t = blockchain.last_block['timestamp']
72     ↗ 記錄自己最後一個區塊的 timestamp
73     # -- ↓ 開始走訪所有其他的節點、下載他們的鏈 ↓ -- #
74     for node in neighbours:
75         response = requests.get(f'{node}/chain')
76         ↗ 對其他節點發出查看它們鏈的 GET 請求 (Requests 套件)
77         if response.status_code == 200:
78             chain = response.json()['chain']          # 取得別人的鏈
79             length = response.json()['length']        # 取得別人的鏈長
80             last_bk_t = response.json()['last_bk_t']
81             ↗ 取得別人鏈中最後一個區塊的時間戳
82             if self.valid_chain(chain): # 先判斷是否為合法鏈
83                 # -- ↓ 再判斷是否滿足 (1) 你的鏈長比我長，或 (2) 我們鏈長相等，
84                 # -- 但你最後一個區塊比我還早產出 ↓ -- #
85                 if (length > max_length) or
86                     (length == max_length and last_bk_t < my_last_bk_t):
87                     max_length = length          # 紀錄新長度
88                     new_chain = chain            # 紀錄新鏈
89     # -- ↓ 若有找到新的合法長鏈 ↓ -- #
90     if new_chain:
91         self.chain = new_chain                  # 將自己的鏈替換成新鏈
92         return True
93     # -- ↓ 沒有找到新合法長鏈 ↓ -- #
94     return False

```



TIPS

請想像一人分飾兩角，待會我們的確會同時運行兩份區塊鏈程式碼。

■ 檢驗鏈的正確性：valid_chain()

上面的程式發出 GET 請求，取得別人的鏈之後，還不能直接拿來比較，要先確定對方的鏈是合法、正確的鏈，接著才進行比較，檢驗正確性的 Method 為 **valid_chain()**：

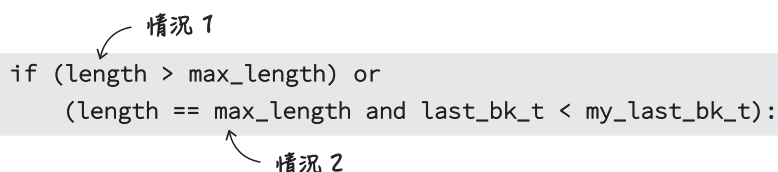
此 Method 新增於 B-10.py 之中

```
092 # -- ↓ 檢驗是否為合法的鏈 ↓ -- #
093 def valid_chain(self, chain):
094     last_block = chain[0]          # 上一個區塊 (0：創世區塊)
095     current_index = 1              # 檢驗索引，1 表示從創世區塊之後的區塊開始檢查
096     while current_index < len(chain): # 開始歷遍整個區塊鏈的區塊
097         block = chain[current_index] # 當前要檢驗的區塊
098         print(f'{last_block}')       # 印出上一個區塊
099         print(f'{block}')            # 印出當前區塊
100         print("\n-----\n")
101         # -- ↓ 1. 確認區塊的雜湊值是否正確 ↓ -- #
102         if block['previous_hash'] != self.hash(last_block):
103             return False
104         # -- ↓ 2. 確認區塊的 nonce 是否正確 ↓ -- #
105         if not self.valid_nonce(last_block['nonce'], block['nonce']):
106             return False
107         # -- ↓ 此區塊驗證成功，換下一個區塊進行驗證 ↓ -- #
108         last_block = block
109         current_index += 1
110     return True # 執行到這裡代表區塊鏈中每個區塊皆驗證成功
```

檢驗區塊鏈中的區塊是否正確，主要就是做 2 個檢驗：

- 1** 確認區塊中儲存的上一個區塊雜湊值是否為正確：將上一個區塊丟進 Method：**hash()** 中，算出區塊的雜湊值；而上一個區塊的雜湊值，應該要存於下一個區塊 (字典) 的鍵：**'previous_hash'** 之中。
- 2** 確認區塊的 **nonce** 是否正確：前面在進行挖礦演算法時有談到，要產生新區塊，必須要找到某個專屬於新區塊的 nonce 數值，而這個 nonce 數值會與上一個區塊的 nonce 有關 (我們設計了前後區塊的 nonce 互相串接後進行雜湊化，這個雜湊化的數值要滿足開頭前 4 個字為 0)。所以當前區塊中的 nonce 與上一個區塊的 nonce 一起丟進 Method：**valid_nonce()** 後，回傳 True 代表當前區塊儲存的 nonce 是正確的。

檢驗正確後，接著在`resolve_conflicts()` 中繼續進行鏈的比較，會出現兩種比較情況：



```
if (length > max_length) or
    (length == max_length and last_bk_t < my_last_bk_t):
```

- **情況 1**：你的鏈比我的鏈長，所以我的鏈被你的取代。
- **情況 2**：我們的鏈一樣長，所以就比較看誰鏈中最後一個區塊產生的時間比較早。

不管滿足哪個情況，都代表我的鏈應該被你的取代，這就是我們的共識演算法，讓節點之間的鏈互相更新。

■ 建立執行共識演算法的路由

有了方法後，接下來我們建立一個路由，讓節點可以使用共識演算法：

此 Method 新增於 B-10.py 之中

```
172 # -- ↓ 路由：執行共識演算法 ↓ -- #
173 @app.route('/nodes/resolve', methods=['GET'])
174 def consensus():
175     replaced = blockchain.resolve_conflicts()
176     if replaced: # 我們節點的鏈被替換成其他節點的長鏈了
177         response = {
178             'message': '我的鏈被取代了',
179             'new_chain': blockchain.chain
180         }
181     else: # 我們節點的鏈為最長鏈，故沒被替換
182         response = {
183             'message': '我的鏈是最長合法鏈，沒被取代',
184             'chain': blockchain.chain
185         }
186
187     return jsonify(response), 200
```

稍後即可對 `/nodes/resolve` 路徑發出 GET 請求，執行共識演算法。

B-5 實戰：運行區塊鏈 - 建立 2 個節點 (村民)

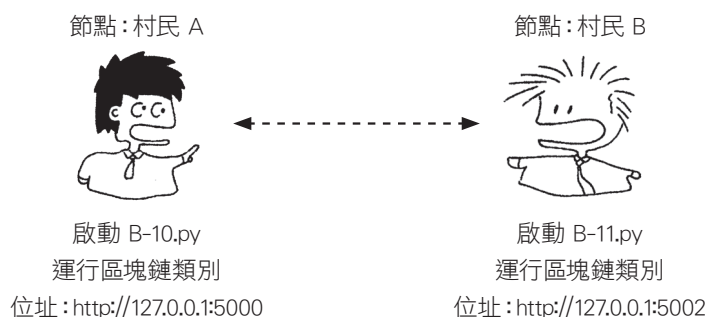
現在我們一人分飾兩角，程式碼 B-10.py 不用動（位址為預設的 `http://127.0.0.1:5000/`），扮演村民 A。

然後複製一份一模一樣的程式碼，但是在啟動網站的部分 `app.run()` 的地方稍作更改：

B-11.py

```
89 if __name__ == '__main__':
90     app.run(host='0.0.0.0', port=5002) # 以 http://127.0.0.1:5002
                                         啟動網站
```

這份程式碼 B-11.py 扮演村民 B，位址為：`http://127.0.0.1:5002`。

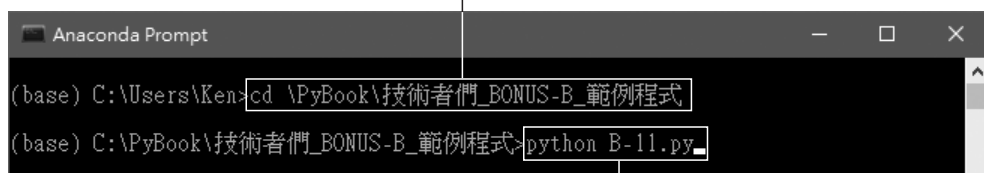


現在我們先將 B-10.py 以 **Spyder** 執行：

```
Python console
Console 1/A
區塊鏈最後一個區塊索引位置：1
* Serving Flask app "B-10" (lazy loading)
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

接著請開啟 **Anaconda Prompt** 命令列視窗來執行 **B-11.py**, 讓這兩個程式碼同時運作：

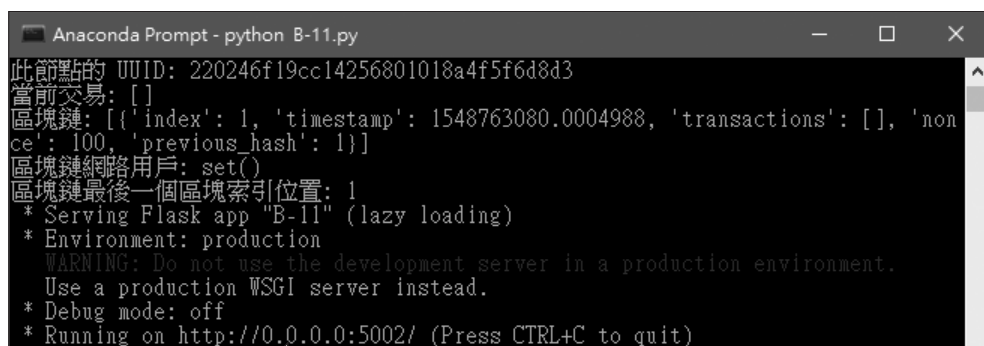
1 使用 cd 指令, 切換到 B-11.py 的目錄位址



```
(base) C:\Users\Ken>cd \PyBook\技術者們_BONUS-B_範例程式
(base) C:\PyBook\技術者們_BONUS-B_範例程式>python B-11.py
```

2 輸入 python B-11.py

輸入後, 即執行程式碼、啟動另一個 Flask 網站 (位址: <http://127.0.0.1:5002>):



```
Anaconda Prompt - python B-11.py
此節點的 UUID: 220246f19cc14256801018a4f5f6d8d3
當前交易: []
區塊鏈: [{'index': 1, 'timestamp': 1548763080.0004988, 'transactions': [], 'nonce': 100, 'previous_hash': 1}]
區塊鏈網路用戶: set()
區塊鏈最後一個區塊索引位置: 1
* Serving Flask app "B-11" (lazy loading)
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:5002/ (Press CTRL+C to quit)
```

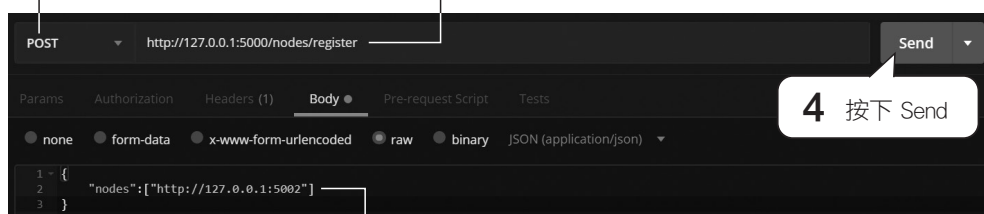
已啟動 <http://127.0.0.1:5002>

現在兩個網站已同時啟動, 請依序以下步驟, 在 Postman 中進行節點之間的區塊鏈的操作：

1 在節點 A 新增節點 B 的位址到區塊鏈鄰居節點清單中

2 選擇 POST 方法

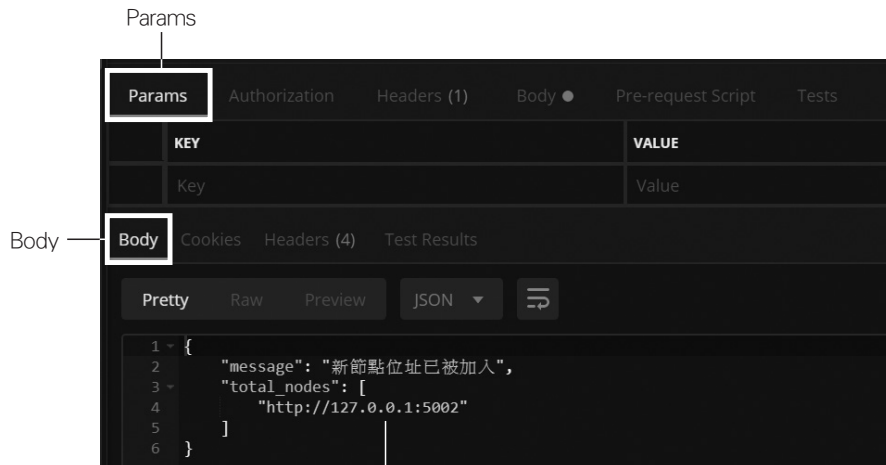
3 輸入請求路徑: <http://127.0.0.1:5000/nodes/register>



4 按下 Send

1 在 Body 分頁中輸入節點 B 的位址

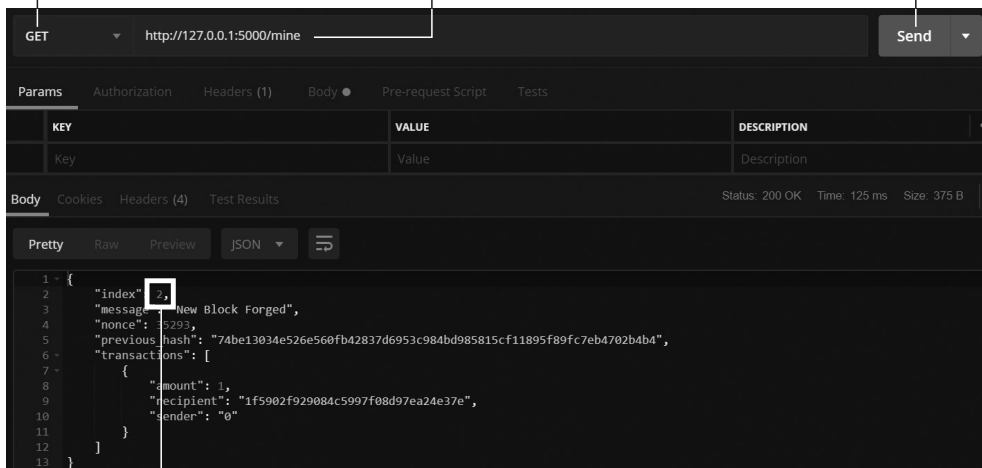
按下 Send 後，可以在 **Params** 分頁中的 **Body** 分頁看到網站回傳的結果：



節點 A 的網站回傳訊息：顯示已加入節點 B 位址

2 在節點 A 進行挖礦，請挖 2 次。

- 1 選擇 GET 方法
- 2 輸入請求路徑 `http://127.0.0.1:5000/mine`
- 3 按下 Send



- 4 可以看到產生一個新區塊，索引為 2

B

區塊鏈實作技術

請直接再按一次 Send, 再挖一次礦：

索引為 3, 現在節點 A 的區塊鏈中有 3 個區塊了

```
1 {
2   "index": 3,
3   "message": "New Block Forged",
4   "nonce": 35089,
5   "previous_hash": "2d11d2999e058f2d7239851ac8cbc9c5aa273ff2e781936334247438649ad14c",
6   "transactions": [
7     {
8       "amount": 1,
9       "recipient": "1f5902f929084c5997f08d97ea24e37e",
10      "sender": "0"
11    }
12  ]
13 }
```

3 在節點 B 新增節點 A 的位址到區塊鏈鄰居節點清單中。

2 選擇 POST 方法 3 輸入請求路徑：http://127.0.0.1:5002/nodes/register 4 按下 Send

1 在 Body 分頁中輸入節點 A 的位址

```
1 {
2   "nodes": ["http://127.0.0.1:5000"]
3 }
```

按下 Send 後, 一樣可以在 Params 分頁中的 Body 分頁看到網站回傳的結果：

Params

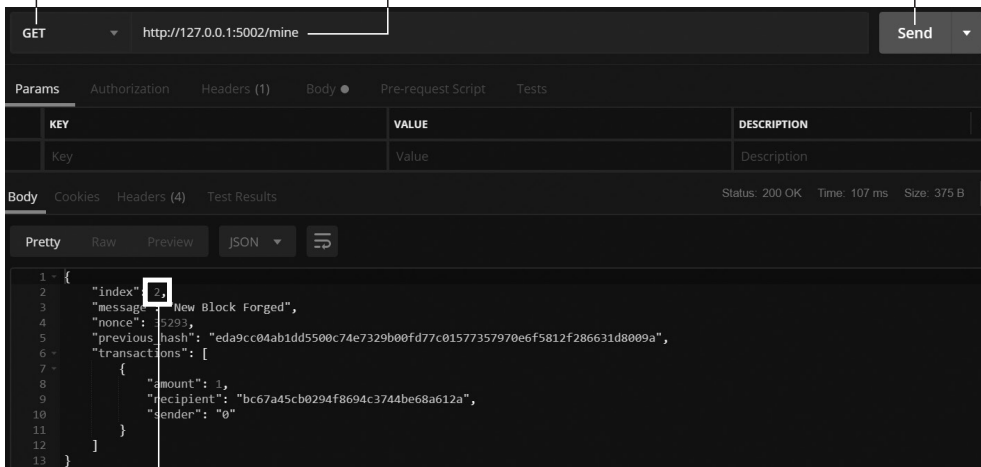
Body

```
1 {
2   "message": "新節點位址已被加入",
3   "total_nodes": [
4     "http://127.0.0.1:5002"
5   ]
6 }
```

節點 B 的網站
回傳訊息：已加入節點 A 位址

4 在節點 B 進行挖礦, 請挖 1 次。

1 選擇 GET 方法 2 輸入請求路徑 http://127.0.0.1:5002/mine 3 按下 Send

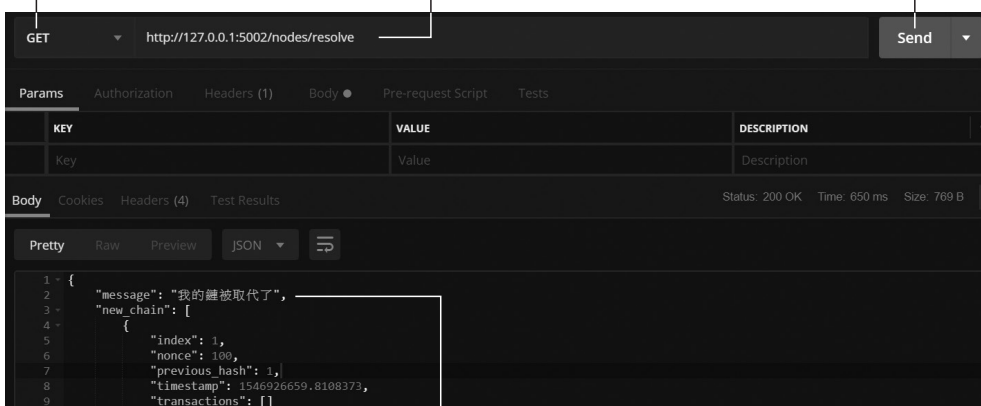


4 可以看到產生一個新區塊, 索引為 2

我們在節點 B 挖一個礦就好了, 現在節點 A 的區塊鏈有 2 個區塊、而節點 B 只有 1 個區塊。

5 在節點 B 執行共識演算法

1 選擇 GET 方法 2 輸入請求路徑 http://127.0.0.1:5002/nodes/resolve 3 按下 Send



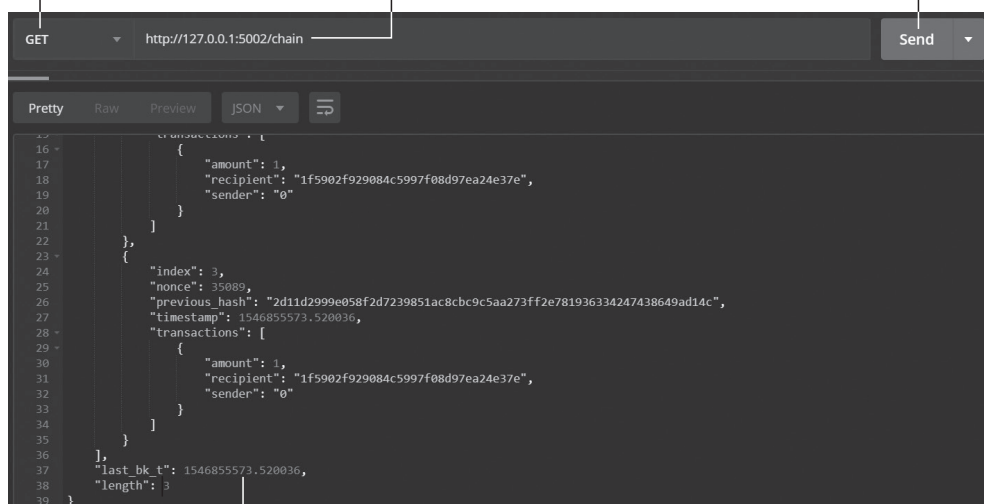
4 可以看到, 網站回傳訊息: 鏈已被取代

B

區塊鏈實作技術

6 查看節點 B 的區塊鏈內容

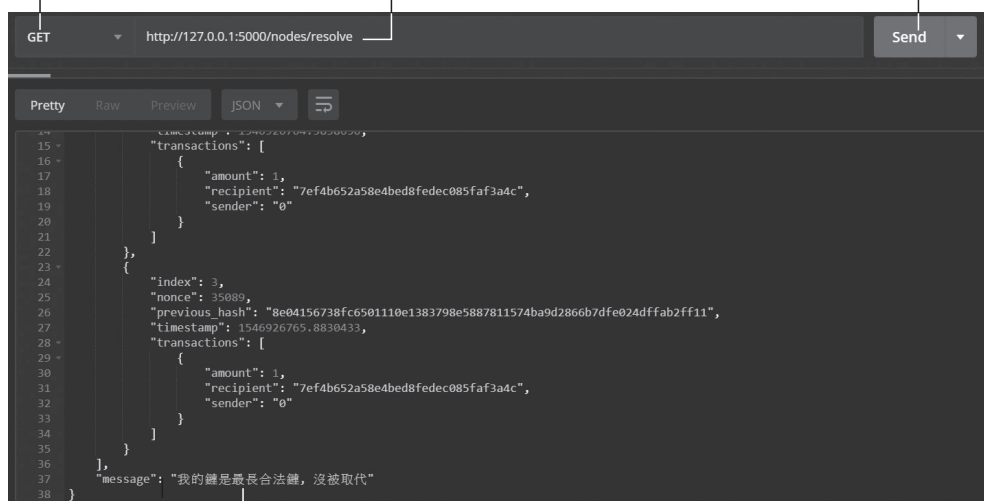
- 1 選擇 GET 方法
- 2 輸入請求路徑 `http://127.0.0.1:5002/chain`
- 3 按下 Send



- 4 區塊有 3 個, 已經變成節點 A 的區塊鏈了!

7 在節點 A 也執行共識演算法看看

- 1 選擇 GET 方法
- 2 輸入請求路徑 `http://127.0.0.1:5000/nodes/resolve`
- 3 按下 Send



- 4 可以看到節點 A 的鏈沒被取代!

節點 A、B 透過共識演算法，達到各節點保持相同的最長合法鏈。這也是為什麼礦工們要比賽看誰挖礦挖得快，才能將自己產生的區塊，加到最長的區塊鏈之中。

本章透過了 Python 打造了一個基礎的區塊鏈網路，讀者可以依據這個類別，再去更進一步的完善區塊鏈的功能，例如替交易 Method 制定更詳細的細節。

當越來越多節點加入一起運行，則此區塊鏈網路就會越來越可靠，虛擬貨幣價值性也會越來越高！

補充學習

A. 51% 攻擊

通常交易記錄被寫上區塊後，會進行所謂的 6 階段確認 (Confirmation)，也就是此區塊的後方要再接著 5 個新區塊，這樣此交易記錄才算安全。防止有心人士製造假交易記錄的區塊。

但除非有心人士擁有此區塊鏈網路 51% 的運算能力，發出所謂的「51% 攻擊」，也就是即使交易記錄在主鏈通過 6 階段確認 (交易確認後，賣方就將商品交給買方)，但這時若擁有 51% 運算力的有心人士想竄改這筆資料，可以在此交易區塊前方建立有異於此交易記錄的新區塊，並在其後方快速建立區塊 (超越主鏈長度)，所以當通過共識演算法後，原本正確的主鏈就被取代掉了。

B. 礦工的秘密

常見的區塊寫入機制是，使用者發出交易記錄後，通常還會附上手續費給礦工，這些交易記錄會先存在礦工的交易池之中，當礦工產生新區塊後，會優先將手續費較高的交易記錄寫到區塊上。