深入 GIL: 如何寫出快速且 thread-safe 的 Python – Grok the GIL: How to write fast and thread-safe Python

2017-05-19 In Python, Python-internals, 翻譯 Tags: Concurrency, GIL, python by louie.lu

本文將會探討 Python 內部的 Global Interpreter Lock，以及學習其如何影響 multi-threaded 程式。

原作者：A. Jesse，Twitter: @jessejiryudavis
原文：Grok the GIL: How to write fast and thread-safe Python

Louie Lu 經作者同意[1][2]翻譯為正體中文。

當我6歲時，我有一個音樂盒。我將他上緊發條，在上面的芭蕾舞者開始繞圈，而盒子內的機關開始敲打，發出「一閃一閃亮晶晶」的聲音。雖然這東西肯定很廉價，不過我喜歡這個音樂盒，而我想要知道他是怎麼運作的。總之，我打開了這個音樂盒，看到了裏面的裝置 —— 一個我拇指大小的金屬圓筒，安裝得當的讓他可以旋轉，透過凸起的牙齒與鋼梳撞擊後發出音符。

在程式設計師的特點中，關於事情如何運作的好奇心是必不可缺的。當我打開我的音樂盒觀看內部時，我展現我可能是一個 —— 如果不是一個頂尖程式設計師 —— 起碼也會是好奇的一個。

這很奇怪，我在對 Global Interpreter Lock (GIL) 誤解的情況下寫了 Python 程式這麼久，因為我還沒有足夠的好奇心來了解他是如何運作的。我遇到很多人跟我有著同樣的猶豫，以及無知。

該是時候把這個黑盒子翹開了。讓我們閱讀 CPython 原始碼來了解什麼是 GIL，為什麼 Python 會有，以及他是如何影響我們撰寫 multi-threaded 程式。我會給出一些範例來讓你了解 GIL。你會學到如何快速寫出 thread-safe 的 Python，以及如何在 threads 與 processes 之間選擇。

(在這邊我們將只專注在 CPython — 不是 Jython，PyPy 或是 IronPython。CPython 是大多數程式設計師使用的 python implmentation。)

Behold, the global interpreter lock

這就是 GIL:

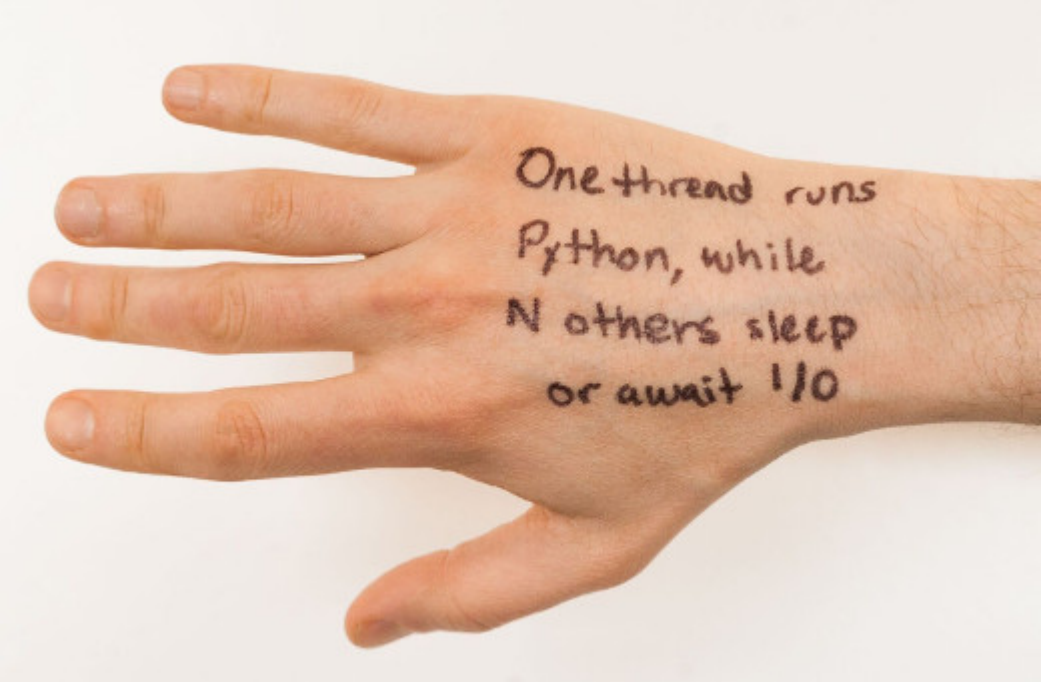
```c
static PyThread_type_lock interpreter_lock = 0; /* This is the GIL */
```

這行程式碼位在 ceval.c，CPython 2.7 的程式碼當中。Guido van Rossum's 的註解「This is the GIL」是在 2003 年的時候加入，不過鎖本身則是在 1997 年時出現在他的第一個 multi-threaded Python 直譯器中。在 Unix systems 中，**PyThread_type_lock** 是 standard C lock, **mutex_t** 的別名。GIL 在 Python 直譯器開始的時候會進行初始化：

```c
void
PyEval_InitThreads(void)
{
    interpreter_lock = PyThread_allocate_lock();
    PyThread_acquire_lock(interpreter_lock);
}
```

所有在直譯器中的 C 程式碼都必須要在執行 Python 時握著這把鎖。Guido 首先以這種方式建構 Python，因為它很簡單。並且，每次從 CPython 移除 GIL的嘗試都讓單執行緒的程式碼效能降低，而無法從多執行緒中獲益。

GIL 對程式所造成的影響簡單到你可以在手背上寫下這個原則：「當有一個執行緒在執行 Python，其他 N 個執行緒都在睡覺或是等待 I/O」。Python 的執行緒們也可以因為 **threading.Lock** 或是其他來自 threading module 的同步物件而等待; 這個狀況下的 thread 我們也可以稱之為「正在睡覺」。



那什麼時候執行緒會切換呢？當一個執行緒正在睡覺或是等待網路 I/O，很有機會其他的執行緒會拿走 GIL 然後開始執行Python code。這就是協調式多任務 (cooperative multitasking)。CPython 同時也有佔先式多任務 (preemptive multitasking): 當一個執行緒沒有直譯超過 1000 個 bytecode 指令 (Python 2)，或是超過 15 ms (Python 3) 的時候，執行緒將會釋放 GIL 來讓其他執行緒使用。你可以想像，這類似於古早年代我們在只有一個 CPU，卻有許多執行緒要跑的狀況下 time slicing。我會更深入的討論到這兩種不同形式的多任務。

協調式多任務 (Cooperative multitasking)

當一個任務開始時，例如說跟網路 I/O 相關，會需要很長一段的等待時間而且不會跑任何的 Python 程式，一個執行緒會將 GIL 給釋放出來，好讓其他的執行緒可以拿走並且工作。這種方式就稱做協調式多任務，而它允許 concurrency；許多的執行緒可以在同一個時間內等待不同的事件。

假設有兩個執行緒想要連接一個 socket:

```
1 import socket
2 import threading
3
4 def do_connect():
5     s = socket.socket()
6     s.connect((‘python.org’, 80))  # drop the GIL
```

```
7
8  for i in range(2):
9      t = threading.Thread(target=do_connect)
10     t.start()
```

在同一個時間點內，只有其中一個執行緒可以執行，不過當執行緒開始連接時，它會將 GIL 釋放好讓其他執行緒可以執行。這代表兩個執行緒可以一起等待他們的 socket connect，這是個好事情，他們可以在等量的時間中做更多的事情。

讓我們撬開黑盒子來看 Python 的執行緒是如何在等待連接建立時把 GIL 給釋放，請看 socketmodule.c:

```c
1  /* s.connect((host, port)) method */
2  static PyObject *
3  sock_connect(PySocketSockObject *s, PyObject *addro)
4  {
5      sock_addr_t addrbuf;
6      int addrlen;
7      int res;
8
9      /* convert (host, port) tuple to C address */
10     getsockaddrarg(s, addro, SAS2SA(&addrbuf), &addrlen);
11
12     Py_BEGIN_ALLOW_THREADS
13     res = connect(s->sock_fd, addr, addrlen);
14     Py_END_ALLOW_THREADS
15
16     /* error handling and so on ….  */
17 }
```

Macro **Py_BEGIN_ALLOW_THREADS** 就是執行緒將 GIL 釋放的地方；他簡單的定義如下：

```c
1  PyThread_release_lock(interpreter_lock);
```

當然，**Py_END_ALLOW_THREADS** 則會重新取得 GIL。一個執行緒可能在這邊被擋住一陣子，等待其他的執行緒釋放 GIL；當 GIL 被釋放後，等待中的執行緒嘗試將 GIL 取回，然後開始執行 Python 程式。簡單來說：當有 N 個執行緒被網路 I/O 阻擋或是等待取得 GIL 時，只有一個執行緒可以執行 Python。

下面會談到如何使用協調式多任務來快速的抓取許多 URL。不過在這個之前，先來比較看看其他的 multitasking 方式。

先佔式多任務 (Preemptive multitasking)

一個 Python 執行緒可以自願的釋放 GIL，不過也可透過搶佔獲得 GIL。

讓我們先喘口氣，講一下 Python 是如何被執行的。你的程式執行被分為兩個階段。第一個階段，你的 Python 程式碼被編譯成一個稱做 *bytecode* 的二進位格式。第二階段，在 Python 直譯器的迴圈中 (前面提到的 ceval.c)，一個名為 **PyEval_EvalFrameEx()** 的函式，會將 bytecode一個一個的讀入並且執行。

當直譯器在執行你的程式的 bytecode 時，會不時的釋放 GIL，而且不會詢問目前正在執行的執行緒，好讓其他的執行緒可以運行：

```c
for (;;) {
    if (--ticker < 0) {
        ticker = check_interval;

        /* Give another thread a chance */
        PyThread_release_lock(interpreter_lock);

        /* Other threads may run now */

        PyThread_acquire_lock(interpreter_lock, 1);
    }

    bytecode = *next_instr++;
    switch (bytecode) {
        /* execute the next instruction … */
    }
}
```

預設情況下 `check_interval` 是 1000 bytecodes。全部的執行緒都跑著相同的程式以及同樣的方式釋放 GIL。在 Python 3 中 GIL 的實作更為複雜，而且 `check_interval` 不是固定數字的 bytecodes，而是 15 ms。不過對你的程式而言，這樣的改動影響並不大。

Thread safety in Python

如果一個執行緒可以在任何時間被迫釋放 GIL，你就必須要確保你的程式碼是 thread-safe。Python 的程式設計師對於 thread safety 的概念與 C 或是 Java 的程式設計師有所差別。這是因為 Python 的許多操作都是*原子化 (atomic)* 的。

舉例而言， 使用 **sort()** 來對一個 list 排序，就是一個 atomic operation。 一個執行緒不能在排序中被打斷，而其他的執行緒不可能看到一個排序到一半的 list，或是看到排序前的舊資料。Atomic operation 簡化了我們的生活，當然也有讓我們驚訝的部份。舉例而言，**+=** 看起來比 **sort()** 簡單，但是 **+=** 不是 atomic 的。我們要如何知道一個操作是 atomic 或不是呢？

考慮有以下的程式：

```
n = 0

def foo():
    global n
    n += 1
```

我們可以用 Python 標準函式庫的 **dis** 模組觀察它的 bytecode：

```
>>> import dis
>>> dis.dis(foo)
LOAD_GLOBAL              0 (n)
LOAD_CONST               1 (1)
INPLACE_ADD
STORE_GLOBAL             0 (n)
```

一行的 **n += 1**，被編譯成 4 個 bytecode，做了 4 個動作：

1. 讀取變數 n 的值放到 stack 上
2. 讀取常數 1 放到 stack 上
3. 將兩個值相加後放到 stack 頂端
4. 將兩數合存回 **n**

還記得每過 1000 個 bytecode，執行緒就會被直譯器中斷，將 GIL 釋放嗎？如果一個 thread 不太幸運，就可能會在第一步到第四步之間被中斷，導致資料無法更新：

```Python
threads = []
for i in range(100):
    t = threading.Thread(target=foo)
    threads.append(t)

for t in threads:
    t.start()

for t in threads:
```

```
    t.join()

print(n)
```

通常你會看到程式印出 *100*，因為 100 個執行緒都把 **n** 增加了 1 次。不過，你有時候會看到 99 或是 98，因為某個執行緒的更新被其他執行緒覆蓋過去了。

所以，就算有 GIL，你還是需要一個鎖來保護共用可變狀態 (shared mutable state):

```python
n = 0
lock = threading.Lock()

def foo():
    global n
    with lock:
        n += 1
```

那如果我們使用 atomic operation，像是 **sort()** 呢？

```python
lst = [4, 1, 3, 2]

def foo():
    lst.sort()
```

這個函式的 bytecode 顯示出 **sort()** 不會被中斷，因為他是 atomic 的：

```python
>>> dis.dis(foo)
LOAD_GLOBAL              0 (lst)
LOAD_ATTR                1 (sort)
CALL_FUNCTION            0
```

這一行程式被編譯成 3 個 bytecode：

1. 讀取 **lst** 的值放到 stack 上
2. 讀取它的 **sort method** 到 stack 上
3. 呼叫 **sort method**

就算這一行 **lst.sort()** 花費了許多步驟，呼叫 **sort** 本身是單一一個 bytecode，這代表在執行完 **sort** 前是沒有機會被中斷而釋放 GIL 的。我們可以得出結論：不需要在 **sort()** 附近上鎖。如果要避免思索哪個操作是不是 atomic，可以遵循一個簡單的原則：永遠在讀

寫共用可變狀態 (shared mutable state) 的附近上鎖。總的來說，使用 **threading.Lock** 的代價在 Python 裏面很低。

雖然 GIL 不代表我們可以遠離鎖，但它代表我們不需要 fine-grained locking。在一個 free-threaded 語言，像是 Java，程式設計師需要盡力的將 shared data 上鎖的時間縮的愈短愈好，來減少 thread contention 以及允許更多的並行性 (allow maximum parallelism)。在 Python 中執行緒不能並行，因此 find-grained locking 並沒有優勢。只要沒有執行緒會在睡眠時持有鎖，做些 I/O 或是其他會釋放 GIL 的操作，你就應該使用最粗糙，最簡單的鎖。總之，其他執行緒沒有辦法並行的執行。

Finishing sooner with concurrency

我賭妳是因為想要透過 multi-threading 來優化程式才來的。如果你的 task 會需要等待許多網路操作的話，雖然 Python 只能在時間內執行一個執行緒，多執行緒仍然能夠幫上忙。這就是 *concurrency*，而執行緒在這樣的場景中能夠運行的很順利。

參考下面的程式碼：

```
1  import threading
2  import requests
3
4  urls = [...]
5
6  def worker():
7      while True:
8          try:
9              url = urls.pop()
10         except IndexError:
11             break  # Done.
12
13         requests.get(url)
14
15 for _ in range(10):
16     t = threading.Thread(target=worker)
17     t.start()
```

如同我們前面所見，這些執行緒會在等待 socket 操作時釋放 GIL，因此他們會比單執行緒跑的還要快。

Parallelism

如果你的 task 只能同時運行 Python 程式碼來降低運行時間該怎麼辦？這種 scaling 稱做 *並行 (parallelism)*，而 GIL 不允許這樣的事情。你必須要使用多個 process — 這會讓程式比起 threading 變得更為複雜而且需要更多的記憶體，不過會因為多核心而受益。

這個範例以 forking 10 個 processes 的方式運行，跑起來會比只有一個 process 快，因為 processes 可以並行的在不同核心執行。只是運行 10 個執行緒不會比只有一個快，因為 Pyhon 同時間只有一個執行緒可以執行：

```python
import os
import sys

nums =[1 for _ in range(1000000)]
chunk_size = len(nums) // 10
readers = []

while nums:
    chunk, nums = nums[:chunk_size], nums[chunk_size:]
    reader, writer = os.pipe()
    if os.fork():
        readers.append(reader)  # Parent.
    else:
        subtotal = 0
        for i in chunk: # Intentionally slow code.
            subtotal += i

        print('subtotal %d' % subtotal)
        os.write(writer, str(subtotal).encode())
        sys.exit(0)

# Parent.
total = 0
for reader in readers:
    subtotal = int(os.read(reader, 1000).decode())
    total += subtotal

print("Total: %d" % total)
```

因為每個 forked 的 process 的 GIL 是分開的，這個程式可以將結果打包並且同時運行多運算。

(Jython 以及 IronPython 有提供 single-process parallelism, 不過他們並沒有支援全部 CPython 的兼容性。PyPy 使用 Software Transactional Memory 在未來可能會變快。如果你好奇的話可以試試這些不同的 Python 直譯器)

結論

現在你已經打開了音樂盒並且看了裏面的機構，你了解了撰寫快速，thread-safe Python 的一切。使用 threads 來處理 concurrent I/O，使用 processes 來處理並行運算。這個原則簡單到你應該不用寫在手背上。

A. Jesse Jiryu Davis will be speaking at PyCon 2017, which will be held May 17-25 in Portland, Oregon. Catch his talk, Grok the GIL: Write Fast and Thread-Safe Python, on Friday, May 19.