УДК 004.422.832:004.432 (045)

Толстікова О.В., к.т.н.,
Мирошниченко І.С.,
Коцюр А.Б.

ЕФЕКТИВНІСТЬ ВИКОРИСТАННЯ ПРОГРАМУВАННЯ АСИНХРОННИХ ДОДАТКІВ МОВОЮ PYTHON

Національний авіаційний університет

tolstikova_alena@mail.ru
izdn_0915@ukr.net

Розглянуто інструменти, які реалізують асинхронне програмування в мові Python і дозволяють підвищити ефективність використання програмування асинхронних додатків. Показано ефективність роботи модуля asyncio (PEP 3156) у порівнянні з класичними співпрограмами

Ключові слова: мова програмування *Python,* модуль *asyncio (PEP 3156),* асинхронне програмування, багатопоточність, обробник події

Вступ

Інтерпретована об'єктно-орієнтована мова *Python* є мовою програмування високого рівня. Структури даних високого рівня разом із динамічною семантикою та динамічним зв'язуванням роблять мову привабливою для швидкої розробки програм. *Python* підтримує модулі та пакети, що сприяє дискретності та повторному використанню коду.

Кожна мова програмування має свої переваги та недоліки, *Python* не є винятком.

Суттєвим недоліком мови програмування *Python* – є низька продуктивність багатопоточних програмних продуктів. Це обумовлено роботою механізму синхронізації потоків *Global Interpreter Lock (GIL)*, що використовується у стандартній реалізації інтерпретатора *Python (CPython)*. Слід зауважити, що більша частина додатків може працювати без створення додаткових потоків. Також, механізм, що оптимізує роботу програми в однопоточному режимі, сповільнює її при виконанні розподілених обчислень. Необхідність написання багатопоточних або асинхронних програм створює суттєву проблему. Надалі буде розглянуто шляхи створення продуктивних та оптимізованих асинхронних програмних продуктів мовою *Python (CPython)* уникаючи обмежень, що накладаються механізмом синхронізації потоків (*GIL*).

Для демонстрації неефективності багатопоточності розглянемо програму, яка буде виводити на друк одиницю 1×10^6 разів.

```
withopen('test1.txt', 'w') asfout:
for i inrange(1000000):
print(file=fout, 1)
```

Виконання цієї програми здійснюється за 0.35 с.

Для порівняння часу виконання розглянемо код програми, яка виводить у файл $0{,}5\times10^6$ одиниць у двох потоках.

```
fromthreadingimportThread
defwriter(filename, n):
withopen(filename, 'w') asfout:
for i inxrange(n):
print(file=fout, 1)
t1 = Thread(target=writer, args=('test2.txt',
500000,))
t2 = Thread(target=writer, args=('test3.txt',
500000,))
t1.start()
t2.start()
```

Час виконання такого коду програми від 0,7с до 7с. Багатопоточна версія працює значно повільніше. Це обумовле-

но динамічним розподіленням *CPU* між потоками, *GIL* та операційною системою.

Отже виконання розподілених обчислень на *Python* слід виконувати в межах одного потоку. Стиль, який використовується для збереження логіки розподіленості у однопоточному програмному коді, є асинхронним.

Асинхронне програмування

Асинхронне програмування відноситься до стилю структурування програми, відповідно до якого виклик деякої одиниці функціональності ініціює дію, яка буде тривати за межами поточного потоку програми. Завдяки оптимізації використання апаратних ресурсів та зручності у написанні програмного коду асинхронне програмування є однією з найпопулярніших парадигм програмування. Слід зауважити, що більшість розроблених додатків є асинхронними об'єктно-орієнтованими програмами.

Для наочності розглянемо це на прикладі графічних інтерфейсів користувача. Коли користувач не взаємодіє з програмою, то вона перебуває у стані очікування. Цей стан можна реалізувати у вигляді постійних перевірок дій користувача (натискання кнопки, наведення курсору, тощо). Такий підхід не є оптимальним, оскільки обчислювальні ресурси витрачаються марно. На сьогодні практично всі *UI*-фреймворки побудовані інакше. Вони реалізують систему обробки подій. Будь-яка дія користувача - це подія, і розробник може прив'язати до неї код - обробник події. Це настільки звичний патерн, що багато розробників навіть не замислюються над принципом його роботи, хоча розуміння цього дуже важливе.

Найбільш поширеною архітектурою системи обробки подій є одно поточна (лише з одним потоком виконання). Однопоточні системи обробки подій практично завжди реалізуються за допомогою черги подій або повідомлень. Принцип роботи однопоточної системи полягає у обробці подій зі стеку по черзі. У потоці можна одночасно обробляти лише одну подію. Таким чином, події додаються в чергу і диспетчер *UI*-фреймворка, який виконується у тому ж потоці, викликає обробники подій у міру потреби. У будь-який момент часу потік може знаходитися або в обробнику події, або в диспетчері, перебуваючи у режимі очікування наступної події. Виникає логічне запитання, яким же чином подія потрапляє в чергу, якщо потік виконання зайнятий обробкою іншої події. Справа в тому, що операційна система має багато потоків, при чому код, який дійсно взаємодіє з користувачем, виконується окремо від програми і лише забезпечує посилання їй повідомлень. Оскільки порядок виконання програмного коду невідомий заздалегідь, ця модель є прикладом асинхронної системи. Обробники подій, з точки зору програми, можуть виконуватися довільно. Але слід зауважити, що в даній моделі обробники подій є неподільними діями. Під час виконання обробника інтерфейс не буде реагувати на дії користувача, оскільки додаток працює в один потік. Виконання складних обчислювальних операцій призводить до «підвисання» інтерфейсу. Причина є в тому, що поки обробник події не повернув управління в диспетчер, наступний обробник подій не буде виконуватися. Для усунення даної проблеми потрібно виконувати обробник цієї події в окремому потоці. Однак в *JavaScript* є лише один потік виконання, а в *Python* проблемою багатопоточних додатків, яка значно їх уповільнює, є *Global Interpreter Lock (GIL)*.

Отже існує два види багатозадачності: витісняюча і кооперативна.

Потоки і процеси використовують витісняючи багатозадачність. Це означає, що операційна система проводить квантування часу і постійно перемикається між різними потоками, зберігаючи і відновлюючи їх контекст виконання.

При використанні кооперативної багатозадачності гілки коду, які виконуються паралельно, самі віддають управління в певні моменти часу. Кооперативна багатозадачність, як спосіб одночасного виконання окремих програм, є застарілою та

не використовується в сучасних операційних системах, однак ідеї, що закладені, виявляються дуже корисними для організації виконання асинхронного коду і дозволяють при грамотному використанні максимально використовувати обчислювальні ресурси в межах одного потоку (а при комбінуванні цього підходу з традиційною багатопоточністю, як в *async*/*await* в *C #*, можна будувати ефективні програми).

Можна побудувати обробник події з безлічі асинхронних функцій зворотного виклику (*callback*-функцій) які управляються загальним циклом подій, як це робиться в *Node.js*, проте такий код складно налагоджувати і підтримувати. Значно спрощують його патерни *Promise* і *Future*, проте *Python* і деякі інші мови програмування підтримують механізм, який дозволяє в даному випадку обійтися без *callback*-функцій. Цей механізм має назву співпрограма.

Співпрограми (*coroutine*) - це компонент програми, узагальнюючий поняття підпрограми, який додатково підтримує безліч вхідних точок (а не одну як підпрограма) і зупинку і продовження виконання зі збереженням певного положення.

Співпрограми в даному випадку зручні тим, що дозволяють писати асинхронний код в синхронному стилі. Розглянемо механізми їх реалізації в мові *Python*.

Реалізація співпрограм мовою Python

Розглянемо класичну реалізацію співпрограм, доступну ще з *Python 2.5*, за допомогою розширених можливостей генераторів (*PEP 342*).

В мові *Python* є механізм, який створено для зручного опису ітераторів. Це - генератори (*generators*).

Функцією-генератором (*generator function*) називається функція, яка може віддавати чергове значення за допомогою ключового слова *yield*, автоматично зберігати і відновлювати свій стан при отриманні наступного значення. При виклику дана функція повертає ітератор, який називається ітератором генератора (*generator iterator*) або об'єктом генератора (*generator object*). Під генератором залежно від контексту розуміють або функцію-генератор, або ітератор генератора.

Приклад:

```
deffibonacci():
    a, b = 0, 1
whileTrue:
yield b
        a, b = b, a + b
```

Даний генератор являє собою нескінченну послідовність чисел Фібоначчі.

Створимо об'єкт генератора:

fibonacci_sequence = fibonacci()

Цей об’єкт буде повертати числа Фібоначчі після кожного виклику.

print (next (fibonacci_sequence))

Вбудована функція *next* викликає метод *__next__ (nextв Python 2)* об'єкта-генератора.

Генератори є окремим випадком співпрограми, так званим *semicoroutines*. На відміну від класичних співпрограм, які можуть передавати управління в довільну співпрограму, генератори можуть передавати його лише в місце виклику методу *__next__*

Саме це і потрібно для реалізації асинхронних функцій: співпрограми повертатимуть управління в цикл подій.

В мові *Python 3* існують підгенератори (*subgenerators*). Об'єкт-генератор, що слідує після пари ключових слів *yieldfrom*, у функції-генераторі делегує доступ до підгенератору, який існує, доки існують значення генератора. Крім того, *yieldfrom* є не оператором, а виразом, результат якого дорівнює тому значенню, яке функція-генератор повертає за допомогою звичайного оператора *return* (воно зберігається у виключенні *StopIteration*, яке виникає при завершенні генератора). Це достатньо для того, щоб реалізувати асинхронне виконання коду на основі циклу подій, і саме так реалізований модуль *asyncio в Python 3.4.*

Однак для реалізації співпрограми в загальному випадку у генераторів в мові *Python* є ще один корисний метод: ***send ()***. При використанні цього метода відправляється в генератор значення, яке дорівнюватиме *yield*-виразу генератора (так, *yield* теж є виразом, а не оператором). За допомогою такого метода можна реалізувати повноцінні співпрограми.

Розглянемо приклад програми, в основі якої лежать дві співпрограми: одна генерує дані (її в такому випадку називають *producer*), а інша - обробляє (*consumer*).

Примітка: тут і далі приклади коду написані на мові *Python 3*.

```
importtime
importrandom
    defsleep(seconds):
    """Призупиняє співпрограму з якої була викликана на задану кількість секунд. """
initial_time = time.time()
whiletime.time() - initial_time<seconds:
yield
    defconsume():
    """Співпрограма обробки данних"""
running_sum = 0
count = 0
whileTrue:
data = yield
    running_sum += data
    count += 1
    print('Gotdata: {}\nTotalcount: {}\nAverage: {}\n'.format(data, count, running_sum / count))
    defproduce(consumer):
    """Співпрограмма видачі даних. Кожні 0.5с генерує випадкове число.
    """
    whileTrue:
    yieldfromsleep(0.5)
    data = random.randint(0, 100)
    consumer.send(data)
    yield
    defmain():
      # Створення обробника даних
    consumer = consume()
       # Запуск співпрограми
    consumer.send(None)
      # Створенння виробника даних
    producer = produce(consumer)
       # Цикл подій (eventloop)
    whileTrue:
    next(producer)
    if__name__ == '__main__':
    main()
```

Функція-генератор *produce* на даному лістингу - це *produced*-співпрограма, *consume* - *consumer*-співпрограма, *sleep*-допоміжна співпрограма, яка використовуються співпрограмою *consume* для припинення виконання свого потоку на заданий час.

На початку роботи програми створюються об'єкти-генератори *consumer* і *producer* і запускається спів програма *consumer*. Її необхідно запустити заздалегідь (або надіславши її значення *None*, або викликавши *next* (*consumer*)), так як єдиним значенням, яке можна відправити в генератор, поки він не запущений, є *None*. Причина у тому, що значення, яке передається за допомогою методу ***send ()***, стає значенням того *yield*-виразу, за допомогою якого генератор повернув управління, отже дане значення неможливо привласнити або зберегти, коли вираз ще не запущений. Далі запускається цикл подій (з єдиною можливим подією: генерація нової порції даних). Оскільки в мові *Python* на рівні мови підтримка повноцінних співпрограм з можливістю передачі управління в задану співпрограму відсутня, використання циклу подій є дуже важливим аспектом роботи програми, але генератори можуть передавати управління лише в функцію що їх викликає. Тому необхідний принцип, за яким здійснювалось управління виконання співпрограм. У цьому випадку таким принципом і є цей цикл.

Розглянемо співпрограму *consume*. Після ініціалізації в нескінченному циклі за допомогою цієї співпрограми забезпечують повторення таких дій: очікування даних за допомогою *yield*-виразу (і, оскільки *yield* віддає управління циклу подій,

в той час, поки дана співпрограма знаходиться в стані очікування, може виконуватися інший код) і їх обробка, тобто збільшення лічильників і виведення на екран отриманого числа, загальної кількості отриманих чисел і їх середнього значення.

Усередині співпрограми *produce* теж знаходиться нескінченний цикл. На кожній ітерації *produse* віддає управління спів програмі *sleep*. Через заданий проміжок часу співпрограма *sleep* віддає управління циклу подій через спів програми *produce*. Далі генерується випадкове число і відправляється об'єкту-генератору *consumer* за допомогою методу *send ()*.

Таким чином, реалізується паралельне виконання двох функцій всередині одного потоку. Однак, на відміну від справжньої багатопоточності, порядок перемикання функцій визначається системою.

Співпрограми дозволяють описувати логіку роботи максимально наближено до багатопоточного коду. Проте в даному випадку рішення зі співпрограмами простіше, елегантніше і не вимагає синхронізації потоків.

Використання модуля asyncio

Модуль *asyncio*, який включений в мову *Python 3.4*, створено у вигляді окремого пакета на *PyPI* для *Python 3.3*. Цей модуль надає всю необхідну інфраструктуру для написання однопоточного конкурентного коду з використанням співпрограми, яка забезпечує неблокування вводу-виводу, мультиплексування вводу-виводу через сокети і інші ресурси, запуску мережевих клієнтів і серверів і т.д.

В основі модуля *asyncio* лежить цикл подій (*eventloop*), який відповідає за:

− створення завдань з співпрограми і *Future* і їх виконання;

− реєстрацію, виконання та скасування відкладених викликів;

− створення клієнтських і серверних транспортів для різних видів комунікації;

− запуск підпроцесів і зв'язку їх з транспортами для взаємодії зі зовнішнім процесом;

− делегування повільних викликів звичайних функцій пулу потоків або пулу процесів.

Співпрограми в модулі *asyncio* - це генератори, які відповідають певним вимогам. До всіх співпрограм повинен бути застосований декоратор *@ asyncio.coroutine.*

Розглянемо приклад двох співпрограм, причому одна з них викликає іншу, яка виробляє якісь витратні обчислення (в даному випадку - припинення свого виконання на деякий час і повертання квадрату числа).

```
importasyncio
 @asyncio.coroutine
deftime_consuming_computation(x):
    print('Computing{0}**2...'.format(x))
    yieldfromasyncio.sleep(1)
    return x ** 2
@asyncio.coroutine
defprocess_data(x):
    result =
    yieldfromtime_consuming_computatio
    n(x)
    print('{0} ** 2 = {1}'.format(x, result))

if__name__ == '__main__':
loop = asyncio.get_event_loop()
loop.run_until_com (process_data(238))
loop.close()
```

Функція *get_event_loop* модуля *asyncio* повертає об'єкт циклу подій і метод *run_until_complete* використовується для запуску співпрограми. Перевагою цього підходу (на відміну від використання звичайних функцій) є те, що під час роботи співпрограми *asyncio.sleep* виконання програми не блокується. Причому інші заплановані для виконання завдання, можуть виконуватися в тому самому потоці.

Переробимо створений вище приклад за допомогою модуля *asyncio*:

```
importasyncio
    importrandom
    importtime
    @asyncio.coroutine
    defconsume():
    """Співпрограма обробки даних"""
    running_sum = 0
count = 0
whileTrue:
data = yieldfromproduce()
running_sum += data
count += 1
print('Gotdata: {}\nTotalcount: {}\nAverage: {}\n'.format(data, count, running_sum / count))

 @asyncio.coroutine
defproduce():
"""Співпрограмма видачі даних."""
yieldfromasyncio.sleep(0.5)
data = random.randint(0, 100)
returndata
defmain():
loop = asyncio.get_event_loop()
loop.run_until_com(consume())
loop.close()
if__name__ == '__main__':
main()
```

Треба звернути увагу, що довелося змінити логіку роботи модуля. Тепер основною є співпрограма *consumer*, а *producer* видає одну порцію даних. Причиною цього є обмеження, що накладаються на співпрограми *asyncio*, які логічно випливають з основного призначення даного модуля, тобто вчинення асинхронного введення-виведення. У більш реальному прикладі аналог співпрограми *producer* міг би, наприклад, отримувати дані з зовнішнього сервера або бази даних.

Висновок

Для вирішення проблеми не оптимізованої багатопоточності у *Python* рекомендовано використовувати асинхронний стиль програмування. Цей стиль дозволяє вирішувати задачі розподілених обчислень у межах одного потоку максимально наближуючи логіку до багатопоточності, проте не успадковуючи проблем пов'язаних із синхронізацією потоків. Розглянуто методи написання програм в асинхронному стилі з використанням співпрограми та стандартного модуля *asincio*.

Використовуючи ці методи можна створювати асинхронні програмні продукти, замість багатопоточних, які сповільнюють *GIL*. Це приводить до більшої оптимізації та покращення якості ПЗ.

Для реалізації асинхронності при створенні додатків мовою *Python* слід використовувати модуль *asyncio*.[1-4]

Список літератури

1. Piligrim M. "Diveintopython 3". – LGPL, 2011.

2. Lutz M. "Programmingpython". - O'ReillyMedia, 2010.

3. Python 3.5.1 documentation URL:https://docs.python.org/3/ (28.03.2016)

4. PythonAsync IO Resources URL: http://asyncio.org/ (28.03.2016)

Статтю подано до редакції 10.02.2016