Модуль multiprocessing в Python, параллельная обработка без GIL

Стандартная библиотека Python3. / Модуль multiprocessing в Python, параллельная обработка без GIL

Параллельная обработка данных на разных ядрах процессора

Пакет multiprocessing поддерживает порождение процессов с использованием API, аналогичного модулю threading.

Модуль многопроцессорной обработки данных предлагает как локальную, так и удаленную параллельную обработку данных, эффективно обходя GIL (глобальную блокировку интерпретатора) и используя ядра процессора вместо потоков. Благодаря этому, этот модуль позволяет программисту полностью **использовать несколько процессоров** на данной машине. Он работает как под Unix, так и под Windows.

Содержание:

Сходство и различия API модулей multiprocessing и threading.

В модуле multiprocessing представлены API, не имеющие аналогов в модуле threading. Ярким примером этого является объект multiprocessing.Pool. Этот объект предлагает удобные средства параллельного выполнения функции для нескольких входных значений, автоматически распределяя их по ядрам процессора.

В следующем примере демонстрируется обычная практика определения таких функций в модуле, чтобы дочерние процессы могли успешно импортировать этот модуль. Этот базовый пример параллелизма данных с использованием пула ядер процессора.

```
import multiprocessing

def worker(x):
    name_proc = multiprocessing.current_process().name
    res = x*x
    print(name_proc, res)
    return res

data = range(3, 7)

with multiprocessing.Pool(2) as pool:
    print('Результаты:')
    print(pool.map(worker, data))

# Результаты:
#  Вверх lWorker-1 9
# ForkPoolWorker-2 16
```

```
# ForkPoolWorker-1 25
# ForkPoolWorker-1 36
# [9, 16, 25, 36]
```

Аналогичный пример с использованием API, аналогичного модулю threading:


```
import multiprocessing

def worker(rear, write):
    while not read.empty():
        name_proc = multiprocessing.current_process().name
        x = read.get()
        res = x*x
        print(name_proc, res)
        write.put(res)
    else:
        read.close()
        write.close()

write = multiprocessing.Queue()
read = multiprocessing.Queue()
[read.put(x) for x in range(3, 7)]

NUM_CORE = 2
procs = []
for i in range(NUM_CORE):
    p = multiprocessing.Process(target=worker, args=(read, write,))
    procs.append(p)
    p.start()

[proc.join() for proc in procs]
print([write.get() for _ in range(write.qsize())])

# Process-1 9
# Process-1 16
# Process-2 25
# Process-1 36
# [9, 16, 36, 25]
```

Контексты и методы запуска процессов на разных ядрах.

В зависимости от платформы модуль multiprocessing поддерживает три способа запуска процесса.

Методы запуска:

spawn:

Родительский процесс запускает новый процесс интерпретатора Python. Дочерний процесс унаследует только те ресурсы, которые необходимы для запуска метода Process.run() объекта multiprocessing.Process. В частности, ненужные файловые дескрипторы и дескрипторы родительского процесса не будут унаследованы. Запуск процесса с использованием этого метода довольно медленный по сравнению с использованием fork или forkserver.

Изменено в Python 3.8: В macOS метод запуска spawn теперь используется по умолчанию. Метод запуска fork следует считать небезопасным, так как он может привести к сбоям подпроцесса.

Доступно в Unix и Windows. По умолчанию в Windows и macOS.

fork:

Родительский процесс использует os.fork() для разветвления интерпретатора Python. Дочерний процесс, когда он начинается, фактически идентичен родительскому процессу. Все ресурсы родительского процесса наследуются дочерним процессом. Обратите внимание, что безопасное разветвление многопоточного процесса проблематично.

Доступно только в Unix. По умолчанию в Unix.

forkserver:

Когда программа запускается и выбирает метод запуска forkserver, запускается процесс сервера. С этого момента всякий раз, когда программе требуется новый процесс, родительский процесс подключается к серверу и запрашивает его ра[Вверх]ние для нового процесса. Процесс сервера является однопоточным, поэтому использование os.fork() безопасно.

Никакие ненужные ресурсы не наследуются.

Доступно на платформах Unix, которые поддерживают передачу дескрипторов файлов по каналам Unix.

В Unix использование методов запуска spawn или forkserver также запускает процесс отслеживания ресурсов, который отслеживает несвязанные именованные системные ресурсы (такие как именованные семафоры или объекты разделяемой памяти), созданные процессами программы. Когда все процессы завершены, трекер ресурсов отсоединяет все оставшиеся отслеживаемые объекты. Обычно их не должно быть, но если процесс был остановлен сигналом, могут быть "*утечки*" ресурсов. Ни семафоры, ни сегменты разделяемой памяти не будут автоматически разъединены до следующей перезагрузки. Это проблематично для обоих объектов, поскольку система допускает только ограниченное количество именованных семафоров, а сегменты разделяемой памяти занимают некоторое пространство в основной памяти.

Чтобы выбрать метод запуска, используете функцию модуля multiprocessing.set_start_method() в предложении if __name__ == '__main__' основного модуля. Функция multiprocessing.set_start_method() не должна использоваться в программе более одного раза.

```
import multiprocessing

def worker(q):
    q.put('hello')

if __name__ == '__main__':
    multiprocessing.set_start_method('spawn')
    q = multiprocessing.Queue()
    p = multiprocessing.Process(target=worker, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

В качестве альтернативы можно использовать функцию multiprocessing.get_context() для получения объекта контекста. Объекты контекста имеют тот же API, что и модуль multiprocessing, и позволяют использовать несколько методов запуска в одной программе.

```
import multiprocessing

def worker(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = multiprocessing.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=worker, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

Обратите внимание, что объекты, относящиеся к одному контексту, могут быть несовместимы с процессами для другого контекста. В частности, блокировки, созданные с использованием контекста fork, не могут быть переданы процессам, запущенным с помощью методов запуска spawn или forkserver.

Библиотека, которая хочет использовать определенный метод запуска, вероятно, должна использовать get_context(), чтобы не мешать выбору пользователя библиотеки.

Предупреждение В настоящее время методы запуска spawn и forkserver не могут использоваться с "замороженными" исполняемыми файлами. То есть с двоичными файлами, созданными такими пакетами, как pyInstaller и cx_Freeze в Unix. Метод запуска fork работает с такими файлами нормально.

Обмен данными между потоками ядер процессора.

При использовании нескольких процессов обычно используется передача сообщений для связи между процессами и избегается необходимости использования каких-либо примитивов синхронизации, таких как блокировки.

Модуль multiprocessing поддерживает два типа каналов связи между процессами.

- Очереди Queues, в собственной реализации.

Вверх

Класс multiprocessing.Queue является почти клоном класса queue.Queue. Очереди безопасны для потоков в разных ядрах процессора.

```
import multiprocessing
```



```
def worker(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = multiprocessing.Queue()
    p = multiprocessing.Process(target=worker, args=(q,))
    p.start()
    print(q.get())
    p.join()

# "[42, None, 'hello']"
```

- Каналы Pipes.

Класс multiprocessing.Pipe() возвращает пару объектов, соединенных каналом, которые по умолчанию является duplex двусторонним.

```
from multiprocessing import

def worker(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = multiprocessing.Pipe()
    p = multiprocessing.Process(target=worker, args=(child_conn,))
    p.start()
    print(parent_conn.recv())
    p.join()

# "[42, None, 'hello']"
```

Два объекта соединения, возвращаемые multiprocessing.Pipe(), представляют два конца канала. Каждый объект подключения имеет методы Pipe.send() - посылает данные в канал и Pipe.recv() - читает данные из канала.

Обратите внимание, что данные в канале могут быть повреждены, если два процесса или потока попытаются читать или записывать в один и тот же конец канала одновременно. Конечно, нет риска повреждения из-за процессов, использующих разные концы канала одновременно.

Синхронизация между процессами на разных ядрах.

Как правило, **примитивы синхронизации не так необходимы в программе**, использующей несколько ядер процессора, как в многопоточной.

Однако, модуль multiprocessing содержит эквиваленты всех примитивов синхронизации из модуля threading. Например, можно использовать блокировку Lock для обеспечения того, что только один процесс печатает на стандартный вывод за раз.

Без использования блокировки вывод различных процессов может все перемешать.

```
import multiprocessing

def worker(lock, i):
    lock.acquire()
    try:
        print('hello world', i)
    finally:
        lock.release()

if __name__ == '__main__':
    lock = multiprocessing.Lock()

    for num in range(10):
        multiprocessing.Process(target=worker, args=(lock, num)).start()
```

Совместное использование состояния между процессами.

Как упоминалось выше, при параллельном программировании обычно **лучше избегать использования общих ресурсов, насколько это возможно**. Это особенно верно при использовании нескольких ядер процессора.

Но если все-же действительно необходимо использование каких-то общих данных, то модуль multiprocessing предоставляет несколько способов сделать это.

- Использование общей памяти *Shared memory*.

Данные могут быть сохранены на карте общей памяти с помощью multiprocessing.Value или multiprocessing.Array.

```
import multiprocessing

def worker(num, arr):
    num.value = 3.1415927
    for i in range(len(arr)):
        arr[i] = -arr[i]

if __name__ == '__main__':
    num = multiprocessing.Value('d', 0.0)
    arr = multiprocessing.Array('i', range(10))

    p = multiprocessing.Process(target=worker, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])

# 3.1415927
# [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Аргументы 'd' и 'i', используемые при создании переменных num и arr, являются кодами типа, который используется модулем array: 'd' указывает на число с плавающей запятой двойной точности, а 'i' указывает на целое число со знаком. Эти общие объекты будут процессными и поточно-ориентированными.

Для большей гибкости в использовании разделяемой памяти можно использовать модуль multiprocessing.sharedctypes, который поддерживает создание произвольных объектов ctypes, выделенных из разделяемой памяти.

- Использование серверного процесса *Server process*.

Объект SyncManager, возвращаемый multiprocessing.Manager(), управляет серверным процессом, который содержит объекты Python и позволяет другим процессам манипулировать ими с помощью прокси-объектов.

Например:

```
import multiprocessing

def worker(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    with multiprocessing.Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))

        p = multiprocessing.Process(target=worker, args=(d, l))
        p.start()
        p.join()

        print(d)
        print(l)

# {0.25: None, 1: '1', '2': 2}
# [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Менеджеры серверных процессов более гибкие, чем использование объектов общей памяти, т. к. могут быть созданы для поддержки произвольных типов объектов. Кроме того, один менеджер может использоваться совместно процессами на разных компьютерах в сети. Однако они медленнее, чем при использовании общей памяти.

Главные принципы программирования для модуля multiprocessing.

Существуют определенные правила и идиомы, которых следует придерживаться при использовании многопроцессорной обработки данных.

Следующие принципы относится ко всем методам запуска.

- **Избегайте общих ресурсов.**
  Насколько это возможно, нужно стараться избегать перемещения больших объемов данных между процессами. Вероятно, лучше придерживаться использования очередей или каналов для связи между процессами, чем использовать примитивы синхронизации более низкого уровня.

- **Picklability.**
  Убедитесь,что аргументы методов прокси-объектов являются упакованы модулем pickle.

- **Потоковая безопасность прокси.**
  Не используйте прокси-объект из более чем одного потока, если вы не защитите его блокировкой. Никогда не возникает проблем с разными процессами, использующими один и тот же прокси.

- **Присоединение к зомби-процессам.**
  В Unix, когда процесс завершается, но к нему не присоединяются, он становится зомби. Их никогда не должно быть очень много, потому что каждый раз, когда запускается новый процесс или вызывается active_children(), все завершенные процессы, которые еще не были присоединены, будут объединены. Также вызов метода Process.is_alive() завершенного процесса присоединится к процессу. Тем не менее, хорошей практикой является явное присоединение ко всем процессам, которые запускаются.

- **Лучше наследовать, чем pickle/unpickle.**
  При использовании методов запуска spawn или forkserver многие типы из multiprocessing должны быть упакованы модулем pickle, чтобы дочерние процессы могли их использовать. Обычно следует избегать отправки общих объектов другим процессам с использованием каналов или очередей.

  В общем необходимо организовать программу так, чтобы процесс, которому требуется доступ к совместно используемому ресурсу, созданному где-то еще, мог унаследовать его от процесса-предка.

- **Избегайте завершения процессов.**
  Использование метода Process.terminate() для остановки процесса может привести к тому, что любые общие ресурсы, такие как блокировки, семафоры, каналы и очереди, в настоящее время используемые процессом, станут сломанными или недоступными для других процессов. Поэтому, вероятно, лучше всего использовать этот метод только для процессов, которые никогда не используют общие ресурсы.

- **Присоединение к процессам, использующим очереди.**
  Имейте в виду, что процесс, который поместил элементы в очередь, будет ждать перед завершением, пока все буферизованные элементы не будут переданы потоком "*питателя*" в нижележащий канал. Дочерний процесс может вызвать метод очереди Queue.cancel_join_thread, чтобы избежать такого поведения.

  Это означает, что всякий раз, когда используется очередь, необходимо убедиться, что все элементы, помещенные в очередь, в конечном итоге будут удалены до присоединения к процессу. В противном случае, никто не может быть уверен, что процессы, поместившие элементы в очередь, завершатся. Помните также, что не демонические процессы будут присоединяться автоматически.

  Вот пример тупиковой ситуации:

```
import multiprocessing

def worker(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = multiprocessing.Queue()
    p = multiprocessing.Process(target=worker, args=(queue,))
    p.start()
```

```
    p.join()    # это тупик
    obj = queue.get()
```

Что бы исправить ситуацию в примере выше, нужно поменять местами последние две строки или просто удалить строку p.join().

- **Явная передача ресурсов дочерним процессам.**
  В Unix, использующем метод запуска fork, дочерний процесс может использовать общий ресурс, созданный в родительском процессе с использованием глобального ресурса. Лучше передать объект в качестве аргумента конструктору дочернего процесса.

  Помимо обеспечения совместимости кода (потенциально) с Windows и другими методами запуска, это также гарантирует, что, пока дочерний процесс все еще жив, объект не будет собираться сборщиком мусора в родительском процессе. Это может быть важно, если какой-то ресурс освобождается при сборке мусора в родительском процессе.

  Так например:

```
import multiprocessing

def worker():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = multiprocessing.Lock()
    for i in range(10):
        multiprocessing.Process(target=worker).start()
```

  следует переписать как:

```
import multiprocessing

def worker(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = multiprocessing.Lock()
    for i in range(10):
        multiprocessing.Process(target=worker, args=(lock,)).start()
```

- **Остерегайтесь замены sys.stdin на файловый объект.**
  Опасность заключается в том, что если несколько процессов вызовут file.close() для этого файлового объекта, то такое поведение может привести к тому, что одни и те же данные будут сброшены в него несколько раз, что приведет к повреждению.

Следующие принципы относится к методы запуска spawn и forkserver.

Есть несколько дополнительных ограничений, которые не применяются к методу запуска fork.

- **Больше picklability.**
  Убедитесь, что все аргументы конструктора Process.__init__() являются picklable. Кроме того, если создается подкласс multiprocessing.Process(), необходимо убедится, что экземпляры будут picklable при вызове метода Process.start().

- **Глобальные переменные.**
  Имейте в виду, если код, выполняемый в дочернем процессе, пытается получить доступ к глобальной переменной, то значение, которое он видит (если оно есть), может не совпадать со значением в родительском процессе во время вызова метода Process.start().

  Однако глобальные переменные, которые являются просто константами уровня модуля, не вызывают проблем.

- **Безопасный импорт основного модуля.**
  Убедитесь, что основной модуль может быть безопасно импортирован новым интерпретатором Python, не вызывая нежелательных побочных эффектов, таких как запуск нового процесса.

  Например, при использовании метода запуска spawn или forkserver, выполняющего следующий модуль, произойдет сбой с исключением RuntimeError:

Вверх

```
multiprocessing import Process
```

```
def worker():
    print('hello')

p = multiprocessing.Process(target=worker)
p.start()
```


Вместо этого следует защитить точку входа программы, используя if __name__ == '__main__':

```
import multiprocessing

def worker():
    print('hello')

if __name__ == '__main__':
    multiprocessing.freeze_support()
    multiprocessing.set_start_method('spawn')
    p = multiprocessing.Process(target=worker)
    p.start()
```

Строку multiprocessing.freeze_support() можно не указывать, если программа будет запускаться в обычном режиме, а не будет заморожена.

Это позволяет вновь созданному интерпретатору Python безопасно импортировать модуль и затем запускать функцию модуля worker().

Подобные ограничения применяются, если пул или менеджер создается в основном модуле.

Содержание раздела:

Вверх