Сообщить об ошибке.

Модультуthread ing в Python, многопоточная обработка данных



<u>Стандартная библиотека Python3.</u> / Модуль threading в Python, многопоточная обработка данных

Параллельная обработка данных на основе потоков

Moдуль threading создает высокоуровневые интерфейсы потоковой передачи данных поверх модуля низкого уровня _thread.

Также смотрите модуль асинхронной очереди queue, для создания заданий для потоков.

<u>Изменено в Python 3.7</u>: раньше этот модуль был необязательным, теперь он доступен всегда.

Конструкция модуля threading основывается на модели потоков в Java. Но там, где Java делает блокировки, а переменные состояния базовым поведением каждого объекта, то в Python они являются отдельными объектами. Класс потоков Python поддерживает подмножество поведения класса потоков Java.

В настоящее время нет приоритетов, нет групп потоков, а потоки не могут быть уничтожены, остановлены, приостановлены, возобновлены или прерваны. Статические методы класса Thread Java при реализации сопоставляются с функциями уровня модуля.

Все описанные методы классов модуля threading выполняются атомарно.

<u>Примечание-1</u>. Чтобы применение многопоточного режима дало ощутимое увеличение производительности, используйте <u>модуль</u> threading, там где встречается много не связанных друг с другом блокирующих операций ввода/вывода. Например, нужно обрабатывать много разрозненных запросов с большой задержкой на ожидание. В режиме "живой очереди" это долго и лучше распараллелить задачу.

<u>Примечание-2</u>. Считать что-то в одном потоке и передавать для дальнейшей обработки другому это не очень здорово, так как возникает лишняя зависимость, которая приводит к снижению производительности, а в случае ошибки приведет к краху всей программы.

Предупреждение. Любой процессор поддерживает определенное количество потоков на ядро, заложенное производителем (обычно 4-6 потоков), при которых он работает оптимально быстро. Нельзя создавать безгранично много потоков. При увеличении числа потоков на величину, большую, чем заложил производитель, программа будет выполняться дольше или вообще поведет себя непредсказуемым образом (вплоть до зависания).

Откажитесь от использования потоков в коде, если нужна хорошая переносимость между разными устройствами. Правильно подобрать число потоков для машины пользователя - трудная задача. Если пишете код под известное "железо", то оптимальное количество потоков можно посмотреть в документации или подобрать экспериментально (тестированием).

Смотрите, пример использования потоков в параллельной обработке файлов.

Как получить значение, возвращаемое потоком Thread в Python?

В документации к модулю threading об этом ничего не сказано, но и так есть масса вариантов решения этой проблемы.

вверх потоки трудятся над какой-то общей задачей, то результаты можно складывать в <u>список</u> или <u>очередь</u>, а по завершении работы получить результаты из соответствующего объекта.

- 2. Если необходимо получать результат работы нескольких потоков, но в той последовательности, в которой стоят задачи (ведь потоки могут возвращать результаты не по порядку), то используйте очередь с приоритетом.
- 3. Если потоки трудятся над разными задачами и результаты работы потоков смешивать нельзя, то возвращаемые значения можно склалывать в словарь, где ключами могут быть имена потоков или простые идентификаторы потоков



апьта

дача, но запущено несколько экземпляров программы и каждая работает в несколько потоков, то 5 оты будет интегральный идентификатор текущего потока threading get native id(). гы программы, потокам необходимо обмениваться результатами, то подойдет та же многопоточная мер обмена информацией между потоками: в примере происходит чтение и обработка файлов из последующей передачей информации в 3-й поток, в котором она записывается в общий файл. этоков нужно получать в реальном времени в основном потоке программы (хотя я не знаю зачем времени см. пункт 5.), то можно в цикле проверять, живы ли потоки, и пока они живы з той же очереди, в которую потоки будут складывать результаты.

<u>Pa</u> <u>ный пример</u>. ir e, random de сть задания while not data.empty(): # получаем задание из очереди с данными task = data.get() # для приличия, умножим хотя бы на 2 res = task * 2# результаты будем возвращать как кортеж, # в котором будет (результат и ID_потока) result.put((res, threading.get_ident())) # имитируем нагрузку $t_sleep = random.uniform(0.5, 2)$ time.sleep(t_sleep) # говорим очереди с данными 'data', # что задание выполнено data.task_done() # заполняем очередь заданиями для потоков # пускай это будет простой список чисел, # которые потоки будут возвращать data = queue.Queue() **for** i **in** range(10, 20): data.put(i) # очередь с возвращаемыми # результатами работы потоков result = queue.Queue() # создаем и запускаем потоки for _ in range(3): # имена потоков будут одинаковыми, что бы можно # было их отличить от основного потока программы thread = threading.Thread(name='worker', target=worker, args=(data, result,)) thread.start() # получаем результаты работы потоков в реальном # времени в основном потоке программы. t_start = time.time() # цикл, пока жив хоть один поток 'worker' while any(th.is_alive() for th in threading.enumerate() if th.name == 'worker'):

!Внимание! очередь с результатами при # работе потоков с разной нагрузкой,

короткие промежутки может быть пустой, ому же мы сразу извлекаем результаты П

В

ПО

```
if not result.empty():
        res, id_thread = result.get()
        # прошедшее время с момента запуска потоков
        tm = round(time.time() - t_start, 2)
                            ead}: результат {res}, время: {tm}')
  РЕКЛАМА
          🔰 ПРОФИЛЬ
#
                            льтат 24, время: 0.62
#
                            льтат 20, время: 0.86
      БОГАТЫЙ ФАСАД
#
                            льтат 26, время: 1.49
   по доступной цене
#
                            льтат 22, время: 1.94
#
                            льтат 28, время: 2.04
                            льтат 30, время: 3.19
#
#
                            льтат 34, время: 3.79
#
                            льтат 32, время: 3.8
#
                            льтат 36, время: 4.59
```

ния потоков в параллельной обработке файлов.

ать каталог на предмет файлов с расширением .txt, а потом обрабатывать их например в 3 чаться в изменении строк и запись измененных данных в другой каталог.

овый каталог и текстовые файлы.

```
# prepare-data.py
import pathlib, random
path = pathlib.Path('.')
# название тестовой директории
test_dir = 'test_dir'
# Путь к тестовой директории
path_dir = path.joinpath(test_dir)
# создаем тестовый директорий
path_dir.mkdir(exist_ok=True)
# количество создаваемых файлов
n_files = 50
# скобочки {} - это шаблон для метода строки
# str.format() туда вставим имя файла
line = "{} - Эту строку будем писать в файл"
if path_dir.is_dir():
    for n in range(n_files):
        # название файла
        f_name = f'file-{n}.txt'
        # путь к файлу
        path_file = path_dir.joinpath(f_name)
        # Генерируем разное количество строк,
        # которые будут писаться в файл
        data = [line.format(f_name) for _ in range(random.randint(5000,15000))]
        # пишем данные в файл
        path_file.write_text('\n'.join(data))
```

Теперь сама программа многопоточной обработки файлов.

Предупреждение: При такой обработке файлов прирост производительности будет незначительным по сравнению с однопоточной обработкой, так как во-первых: 3 потока создают дополнительную загрузку файловой системы (одновременное чтение/запись 3-х файлов), следовательно файловая система будет работать медленнее, чем при однопоточной. И, во вторых: GIL еще ни кто не отменял.

```
# новый файл на запись
        with open(job, 'r') as fr, open(file_write, 'w') as fw:
            # дописываем имя файла
            fw.write(f'\n\n========> {file_write}\n\n')
                            построчно
  РЕКЛАМА
             АПЬТА
             профиль
                            заменим букву у на О
                            replace('y', '0')
      БОГАТЫЙ ФАСАД
                            ненные данные
    по доступной цене
                            e)
                            то задача выполнена
рą
#
t٤
#
                             t_dir)
рą
#
test_dir_modified = 'test_dir_modified'
path_dir_modified = path.joinpath(test_dir_modified)
path_dir_modified.mkdir(exist_ok=True)
# создаем и заполняем очередь именами файлов
que = queue.Queue()
for file in list_files:
    que.put(file)
if que.qsize():
    # Создаем и запускаем потоки
    n_{thead} = 3
    for _ in range(n_thead):
        th = threading.Thread(target=worker, args=(que,), daemon=True)
        th.start()
    # Блокируем дальнейшее выполнение
    # программы до тех пор пока потоки
    # не обслужат все элементы очереди
    que.join()
else:
    print('Файлы не найдены.')
```

Содержание раздела:

- КРАТКИЙ ОБЗОР МАТЕРИАЛА.
- Получение общих сведений о потоках, модуль threading
- Класс Thread() модуля threading
- <u>Knacc local() модуля threading</u>
- <u>Knacc Event() модуля threading</u>
- <u>Knacc Lock() модуля threading</u>
- <u>Knacc RLock() модуля threading</u>
- Класс Condition() модуля threading
- <u>Knacc Semaphore() модуля threading</u>
- <u>Kлacc Timer() модуля threading</u>
- Класс Barrier() модуля threading
- <u>Протокол управления контекстом в модуле threading</u>
- <u>Трассировка и профилирование потоков модулем threading</u>
- <u>Как перезапускать потоки?</u>

BBepx thon.ru™, 2023 г.

(Внимание! При копировании материала ссылка на источник обязательна)

@docs_python_ru