Урок Асинхрон

Введение в асинхронное программирование.

1. **Введение**
2. **Асинхронная программа.**
3. **Немного терминов.**
4. **Где асинхронность применяется в программировании?**
5. **Примеры асинхронных программ.**

Аннотация

Сегодня мы узнаем, что такое асинхронное программирование, чем оно отличается от синхронного, как писать асинхронные программы на python.

Представьте, что вы готовите угощение из трёх блюд, которое содержит следующее:

- закуску, которая потребует 2 минут на подготовку и 3 минут приготовления-ожидания
- основное блюдо, занимающее 5 минут подготовительных работ и 10 минут приготовления-ожидания
- десерт, отнимающий 3 минут на предварительные мероприятия и 5 минут приготовления-ожидания

Ваша задача - приготовить эти блюда за наименьшее время. Например, если мы будем все делать последовательно, то на закуску у нас уйдёт 5 минут, на основное блюдо - 15 минут и, наконец, десерт - ещё 8 минут. Всего потребуется 28 минут.

Как же можно уменьшить необходимое время? Идея в том, что пока вы ждете приготовления одного блюда, можно заняться подготовкой к другому. Таким образом временные промежутки ожидания одного блюда и подготовки к приготовлению другого будут перекрываться.

Например, при помощи следующих шагов можно улучшить результат:

1. Подготовка закуски: 2 минуты.
2. Подготовка основного блюда в то время, пока готовится закуска: 5 минут. Приготовление закуски завершится на этом этапе.

3. Подготовка и приготовление десерта пока вы дожидаетесь завершения приготовления основного блюда: 8 минут. Десерт уже будет готов, а основному блюду потребуется ещё 2 минуты до окончательной готовности.

4. Ожидание готовности основного блюда: 2 минуты.

Итого 17 минут на все вместо 28 в предыдущем варианте.

Очевидно, что существует более одного способа уменьшения времени на приготовление. В этом примере можно уложиться в 15 минут.

Первый способ - это **синхронный** режим. Все действия выполняются последовательно друг за другом. Пока не будет окончено выполнение одного действие, другое начинать нельзя. Все действия в этом случае являются блокирующими.

Второй способ - это **асинхронный** режим. Некоторые действия блокирующие - подготовка ингредиентов. Некоторые неблокирующие, например, ожидание приготовления блюда, в это время можно заняться чем-то еще, например, подготовкой ингредиентов для следующего блюда. При этом не привлекается еще один человек, т.е. используется только один поток выполнения.

Давайте теперь, напишем программу, которая моделирует приготовление блюд по алгоритмам, описанным выше. Для удобства в программе одна минута реального времени будет соответствовать одной секунде времени работы программы, чтобы не надо было ждать 28 минут, пока все "блюда" приготовятся.

Программа для первого **синхронного** варианта может выглядеть так:

```
import time
from datetime import datetime

# в нашей программе одна минута реального времени будет соответствовать одной секунде врем
# если установить коэффициент в 0.5, то половине секунды времени работы программы
COEFF = 1


def dish(num, prepare, wait):
    """
    функция имитирует приготовление одного блюда
    :param num: номер блюда по порядку
    :param prepare: сколько минут на подготовку
    :param wait: сколько минут на ожидание готовности
    :return:
    """
    print(f"start {datetime.now().strftime('%HH:%MM:%SS')} prepare dish {num} {prepare} m
    time.sleep(COEFF * prepare)
    print(f"start {datetime.now().strftime('%HH:%MM:%SS')} wait dish {num} {wait} min")
    time.sleep(COEFF * wait)
    print(f"{datetime.now().strftime('%HH:%MM:%SS')} dish {num} is ready")


def main():
    """
```

```
    функция запускает "приготовление" блюд
    :return:
    """
    dish(1, 2, 3)
    dish(2, 5, 10)
    dish(3, 3, 5)


if __name__ == '__main__':
    t0 = time.time()  # запоминаем время начала работы
    main()  # запускаем приготовление
    delta = int((time.time() - t0) / COEFF)  # считаем затраченное время
    print(f"{datetime.now().strftime('%HH:%MM:%SS')} It took {delta} min")
```

1. Асинхронная программа.

Теперь напишем **асинхронный** вариант с использованием стандартной библиотеки asyncio:

```
import os
import time
import asyncio
from datetime import datetime

COEFF = 1


async def dish(num, prepare, wait):
    print(f"start {datetime.now().strftime('%HH:%MM:%SS')} prepare dish {num} {prepare} m
    time.sleep(COEFF * prepare)
    print(f"start {datetime.now().strftime('%HH:%MM:%SS')} wait dish {num} {wait} min")
    await asyncio.sleep(COEFF * wait)
    print(f"{datetime.now().strftime('%HH:%MM:%SS')} dish {num} is ready")


async def main():
    tasks = [
        asyncio.create_task(dish(1, 2, 3)),
        asyncio.create_task(dish(2, 5, 10)),
        asyncio.create_task(dish(3, 3, 5)),
    ]
    await asyncio.gather(*tasks)


if __name__ == '__main__':
    t0 = time.time()  # запоминаем время начала работы
    if os.name == 'nt':
        asyncio.set_event_loop_policy(asyncio.WindowsSelectorEventLoopPolicy())
    asyncio.run(main()) # запускаем асинхронное приготовление
```

```
    delta = int((time.time() - t0) / COEFF)  # считаем затраченное время
    print(f"{datetime.now().strftime('%HH:%MM:%SS')} It took {delta} min")
```

Перед разбором примера несколько пояснений:

1. Функции, которые не являются блокирующими и могут выполняться асинхронно, называются корутинами **(coroutines)**. Для описания таких функций, начиная с Python 3.5, используют ключевое слово `async`:

```
async def my_coro():
    pass
```

2. Для выполнения неблокирующего вызова корутины используется ключевое слово `await`. Оно, как бы, говорит, что здесь мы начинаем ждать, и пока мы ждем выполнение программы может быть передано куда-то еще. Как только ожидание закончится, выполнение функции (корутины) будет продолжено до ее окончания или до следующей директивы await.
3. `asyncio.create_task()` - создает задачу на основе корутины. В этом примере мы создаем три задачи для приготовления трех блюд и помещаем их в список.
4. `asyncio.gather()` - "собирает" все асинхронные задачи, которые необходимо выполнить, запускает их и дожидается (`await`) их выполнения.
5. `asyncio.run()` - запускает корутину на выполнение.

В приведенном примере есть две корутины

1. `async def dish(num, prepare, wait)` - приготовление блюда номер `num`, `prepare` минут на подготовку ингредиентов, `wait` минут на ожидание готовности. Внутри корутины использован блокирующий вызов (т.е. выполнение программы "замораживается") `time.sleep()` - приготовление ингредиентов, и неблокирующий асинхронный вызов `await asyncio.sleep()`. Когда выполнение корутины дойдет до `await asyncio.sleep()`, то ее работа будет временно приостановлена, а управление передано следующей на очереди задаче. Выполнение корутины будет возобновлено, по возможности, как можно раньше, но не ранее завершения ожидания.
2. `async def main()` - это корутина предназначение для составления списка задач, которые надо выполнить

```
tasks = [
        asyncio.create_task(dish(1, 2, 3)),
        asyncio.create_task(dish(2, 5, 10)),
        asyncio.create_task(dish(3, 3, 5)),
    ]
```

3. `asyncio.run(main())` - Точка входа в программу. Создает и запускает цикл обработки событий. Принимает в качестве параметра корутину верхнего уровня.

Обратите внимание на строки:

```
if os.name == 'nt':
        asyncio.set_event_loop_policy(asyncio.WindowsSelectorEventLoopPolicy())
```

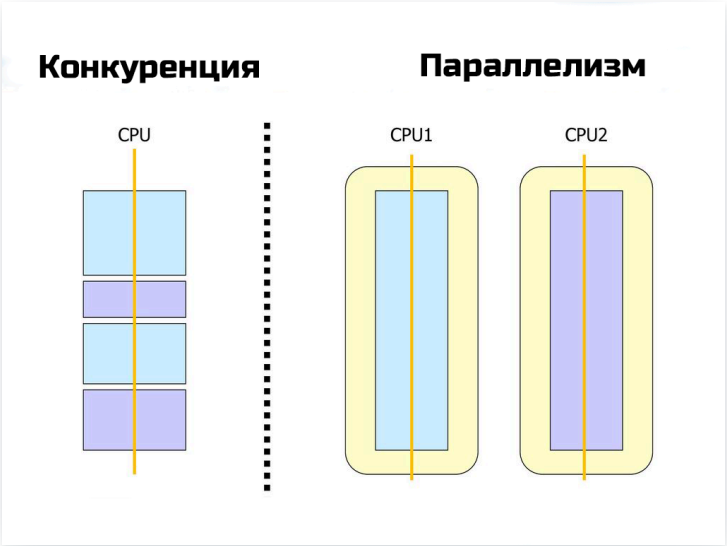
Дело в том, что в разных операционных системах, асинхронные вызовы могут быть реализованы неодинаково. Здесь мы ставим "костыль" для операционной системы Windows.

2. Немного терминов.

Когда говорят об асинхронности, обычно используют три близких понятия. Это - **конкурентность** (concurrency), **параллелизм** (parallel execution) и **многопоточность** (multithreading). Все они связаны с одновременным выполнением задач, однако, это не одно и то же.

Конкурентность.
Понятие конкурентного исполнения самое общее. Оно означает, что множество задач выполняются в одно время. Можно сказать, что в программе есть несколько логических потоков – по одному на каждую задачу. При этом потоки могут физически выполняться одновременно, но это не обязательно. Задачи при этом не должны быть связаны друг с другом, и не имеет значения, какая из них завершится раньше, а какая позже.

Параллелизм.
Параллельное исполнение используют для разделения одной задачи на части для ускорения вычислений. Например, нужно сделать цветное изображение черно-белым. Обработка верхней половины не отличается от обработки нижней. Следовательно, можно разделить эту задачу на две части и раздать их разным потокам, чтобы ускорить выполнение в два раза. Наличие двух физических потоков здесь принципиально важно, так как на компьютере с одним вычислительным устройством (процессорным ядром) такой прием провести невозможно.



Многопоточность.
Здесь поток является абстракцией, под которой может скрываться и отдельное ядро процессора, и **тред ОС**. Некоторые языки даже имеют собственные объекты потоков. Таким образом, эта концепция может иметь принципиально разную реализацию.

Асинхронность.
Идея асинхронного выполнения заключается в том, что начало и конец одной операции происходят в разное время в разных частях кода. Чтобы получить результат, необходимо подождать, причем время

ожидания непредсказуемо. Как правило, асинхронные задачи выполняются в одном потоке. Такой режим еще называют **"кооперативная однопоточная многозадачность".** Т.е. асинхронность - это частный случай конкурентности. Именно эта идея и была реализована в разобранном выше примере при помощи библиотеки asyncio.

3. Где асинхронность применяется в программировании?

Асинхронность больше всего подходит для таких сценариев:

1. Программа выполняется слишком долго.
2. Причина задержки — не вычисления, а ожидания ввода или вывода.
3. Задачи, которые включают несколько одновременных операций ввода и вывода.

   Это могут быть:

   - Парсеры (обработчики),
   - Сетевые сервисы (получение и отправка данных)
   - Работа с данными (запросы в базы данных)

4. Асинхронный подход не применим в задачах, которые сильно нагружают процессор, например, расчет или проверка чисел на простоту. Здесь может спасти многопоточность.

Далее мы разберем еще несколько примеров асинхронных программ.

4. Примеры асинхронных программ.

Worker Pool.
Необходимо создать набор (pool) функций (workers), которые будут заниматься тем, что получают данные и выполняют какую-либо их обработку. При этом от момента запроса до момента получения данных может пройти достаточно большое время (до нескольких миллисекунд).

Синхронное решение.

```
import time
from random import randint

ITER_NUM = 10  # количество итераций в каждом воркере
COROUT_NUM = 5  # необходимое количество воркеров


def do_some_work(i):
    for j in range(ITER_NUM):
        print(format_work(i, j))
        time.sleep(0.01 * randint(1, 10))


def main():
    for i in range(COROUT_NUM):
```

```
        do_some_work(i)


def format_work(i, j):
    return i * ' ' + "▉" + (COROUT_NUM - i) * ' ' + f'cr {i} iter {j} ' + "■" * j


if __name__ == '__main__':
    main()
```

В программе создается некоторое количество (`COROUT_NUM`) воркеров, каждый из которых выполняет некоторое количество итераций (`ITER_NUM`). Каждая итерация получения данных занимает случайное время. В нашем примере единственная работа, которую делает воркер - выводит информацию о себе и о своих итерациях.

Запустим код:

```
▉     cr 0 iter 0
▉     cr 0 iter 1 ■
▉     cr 0 iter 2 ■■
▉     cr 0 iter 3 ■■■
▉     cr 0 iter 4 ■■■■
▉     cr 0 iter 5 ■■■■■
▉     cr 0 iter 6 ■■■■■■
▉     cr 0 iter 7 ■■■■■■■
▉     cr 0 iter 8 ■■■■■■■■
▉     cr 0 iter 9 ■■■■■■■■■
 ▉    cr 1 iter 0
 ▉    cr 1 iter 1 ■
 ▉    cr 1 iter 2 ■■
 ▉    cr 1 iter 3 ■■■
 ▉    cr 1 iter 4 ■■■■
 ▉    cr 1 iter 5 ■■■■■
 ▉    cr 1 iter 6 ■■■■■■
 ▉    cr 1 iter 7 ■■■■■■■
 ▉    cr 1 iter 8 ■■■■■■■■
 ▉    cr 1 iter 9 ■■■■■■■■■
```

Поскольку решение синхронное, то все воркеры и итерации выполняются строго друг за другом.

Асинхронное решение.

```
import asyncio
import os
from random import randint

ITER_NUM = 10
COROUT_NUM = 5
```

```
async def do_some_work(i):
    for j in range(ITER_NUM):
        print(format_work(i, j))
        await asyncio.sleep(0.01 * randint(1, 10))


async def main():
    tasks = []
    for i in range(COROUT_NUM):
        tasks.append(asyncio.create_task(do_some_work(i)))
    await asyncio.gather(*tasks)


def format_work(i, j):
    return i * ' ' + "▉" + (COROUT_NUM - i) * ' ' + f'cr {i} iter {j} ' + "■" * j


if __name__ == '__main__':
    if os.name == 'nt':
        asyncio.set_event_loop_policy(asyncio.WindowsSelectorEventLoopPolicy())
    asyncio.run(main())
```

Здесь функция `async def do_some_work(i)` является асинхронной. В ней выполняется блокирующая функция - `print`, и асинхронная `await asyncio.sleep()`.

Запустим код:

```
▉     cr 0 iter 0
 ▉    cr 1 iter 0
  ▉   cr 2 iter 0
   ▉  cr 3 iter 0
    ▉ cr 4 iter 0
 ▉    cr 1 iter 1 ■
    ▉ cr 4 iter 1 ■
  ▉   cr 2 iter 1 ■
 ▉    cr 1 iter 2 ■■
▉     cr 0 iter 1 ■
  ▉   cr 2 iter 2 ■■
   ▉  cr 3 iter 1 ■
▉     cr 0 iter 2 ■■
▉     cr 0 iter 3 ■■■
    ▉ cr 4 iter 2 ■■
 ▉    cr 1 iter 3 ■■■
   ▉  cr 3 iter 2 ■■
```

Здесь очень хорошо видно, что сначала выполняется 0-я итерация каждого из запущенных воркеров, а затем каждая из их итераций выполняется в зависимости от времени, которое надо подождать (а оно

случайное).

Загрузка файлов из сети интернет и их локальное сохранение.
Мы будем получать случайные картинки из интернета и сохранять их на локальный диск. Скачивать будем с сайта **loremflickr.com**. Каждый запрос get, отправленный по этому адресу, будет перенаправлен на случайное изображение размером 640X480. Сама операция получения изображения является "узким горлышком". В момент получения информации от сервера, синхронная программа будет блокировать и просто "ждать". Программа сохраняет файлы в папку img, не забудьте ее создать.

```
import requests
from time import time


def get_file(i, url):
    print(f"getting file number {i}")
    r = requests.get(url, allow_redirects=True)
    write_file(i, r)


def write_file(i, response):
    print(f"writing file number {i}")
    filename = response.url.split('/')[-1]
    with open(f'./img/{filename}', 'wb') as file:
        file.write(response.content)


def main():
    url = 'https://loremflickr.com/640/480'
    for i in range(30):
        get_file(i, url)


if __name__ == '__main__':
    t0 = time()
    main()
    print(f'{time() - t0} seconds')
```

Функция `def get_file(url)` получает ссылку и выполняет **get** запрос. Результат выполнения запроса передается в функцию `write_file()`, которая сохраняет файл на диск.
Пример вывода и время работы программы:

```
getting file number 0
writing file number 0
getting file number 1
writing file number 1
getting file number 2
writing file number 2
getting file number 3
writing file number 3
```

```
getting file number 4
writing file number 4
getting file number 5
    ..............
    ..............
getting file number 27
writing file number 27
getting file number 28
writing file number 28
getting file number 29
writing file number 29
21.349416255950928 seconds
```

Файлы последовательно скачиваются и записываются на диск. Синхронная программа обработала 30 файлов за 21.3 секунды.

Перепишем эту программу асинхронно. Для этого применим библиотеку `pip install aiohttp`. Она позволяет вызывать http методы асинхронно. **Документация и примеры**.

```
import os
from time import time
import asyncio
import aiohttp


async def get_file(i, url, session):
    print(f"getting file number {i}")
    response = await session.get(url, allow_redirects=True)
    await write_file(i, response)
    response.close()


async def write_file(i, response):
    print(f"writing file number {i}")
    filename = response.url.name
    data = await response.read()
    with open(f'./img/{filename}', 'wb') as file:
        file.write(data)


async def main():
    url = 'https://loremflickr.com/640/480'
    tasks = []
    session = aiohttp.ClientSession()
    for i in range(30):
        task = asyncio.create_task(get_file(i, url, session))
        tasks.append(task)
    await asyncio.gather(*tasks)
    await session.close()
```

```
if __name__ == '__main__':
    t0 = time()
    if os.name == 'nt':
        asyncio.set_event_loop_policy(asyncio.WindowsSelectorEventLoopPolicy())
    asyncio.run(main())
    print(f'{time() - t0} seconds')
```

Корутина `async def get_file(url, session)` получает ссылку и сессию, в рамках которой надо выполнить запрос. Можно было бы создавать сессию на каждый запрос (как это и было в синхронном режиме), но это займет больше времени. В корутине асинхронно запускается **get** запрос, и пока программа ждет на него ответа, отправляет запросы на получения остальных файлов. По мере получения данных асинхронно вызывается функция `await write_file()`, которая асинхронно "вычитывает" данные из полученного ответа и сохраняет их в файл на диск. Операция сохранения файла уже блокирующая. Так же по окончании работы с ответом надо его закрыть `response.close()` и после получения всех данных надо закрыть сессию `await session.close()`. Чтобы не писать каждый раз закрытие ответа и сессии, можно использовать асинхронный контекстный менеджер async with, по аналогии с синхронным with.

```
async def get_file(i, url, session):
    print(f"getting file number {i}")
    async with session.get(url, allow_redirects=True) as response:
        await write_file(i, response)

async def main():
    url = 'https://loremflickr.com/640/480'
    tasks = []
    async with aiohttp.ClientSession() as session:
        for i in range(30):
            task = asyncio.create_task(get_file(i, url, session))
            tasks.append(task)
        await asyncio.gather(*tasks)
```

Пример вывода и время работы программы:

```
getting file number 0
getting file number 1
getting file number 2
getting file number 3
getting file number 4
getting file number 5
getting file number 6
getting file number 7
getting file number 8
        ..............
        ..............
```

```
getting file number 28
getting file number 29
writing file number 3
writing file number 22
writing file number 29
writing file number 1
writing file number 5
writing file number 24
        ..........
        ..........
writing file number 12
writing file number 28
1.3730313777923584 seconds
```

Сначала все файлы были запущены на скачивание. Затем, по мере получения файлов, они записываются на диск. 30 файлов были обработаны за 1.37 секунды.

Получение результатов работы из асинхронной функции.

```
import os
import aiohttp
import asyncio

urls = ['http://www.google.com', 'http://www.yandex.ru', 'http://www.python.org', 'http:/,


async def get_url(url, session):
    print(f'GETTING URL: {url}')
    async with session.get(url) as resp:
        print(f'URL {url} RECEIVED STATUS: {resp.status}')
        text = await resp.text()
        print(f'URL {url} LEN TEXT: {len(text)}')
        return url, text


async def main():
    counter = 0
    async with aiohttp.ClientSession() as session:
        tasks = [get_url(url, session) for url in urls]
        for future in asyncio.as_completed(tasks):
            counter += 1
            res_url, res_text = await future
            print(f"the {counter} len result for url {res_url} was: {len(res_text)}")


if __name__ == '__main__':
    if os.name == 'nt':
```

```
        asyncio.set_event_loop_policy(asyncio.WindowsSelectorEventLoopPolicy())
    asyncio.run(main())
```

Для получения результатов работы из асинхронной функции используется `asyncio.as_completed()`. Эта функция принимает корутины, преобразует их в задачи и запускает на выполнение, она заменяет `asyncio.create_task()` и `asyncio.gather()`. В качестве результатов работы возвращает итератор, содержащий объекты типа `asyncio.Future`. Для каждого такого объекта надо вызвать await и он вернет результат работы корутины.