1. Теория
2. Практика
3. Сопрограмму можно рассматривать как обычную функцию Python. Слово **async** определяет сопрограмму.

import asyncio

async def cor(number):

return number + 1

result = cor(5)

print(result, type(result))

-------------------------------

<coroutine object cor at 0x000001fa1b5119c0> <class 'coroutine'>

Код выше возвращает объект сопрограммы, а не результат.

1. Сопрограмма наделена сверхспособностью - приостанавливаться, встретив операцию, для выполнения которой нужно заметное время. По завершении такой длительной операции сопрограмму можно “пробудить”, после чего она продолжит выполнение. Пока приостановленная сопрограмма ждет завершения операции, мы можем выполнять другой код. Такое выполнение другого кода во время ожидания и обеспечивает конкурентность внутри приложения. Можно также одновременно выполнять несколько длительных операций, что еще больше повышает производительность приложения.

Сопрограммы не выполняются, если их вызвать напрямую. Вместо этого возвращается объект сопрограммы, который будет выполнен позже. Чтобы выполнить сопрограмму, мы должны явно передать ее **циклу событий**. Можно воспользоваться функцией **asyncio.run():**

import asyncio

import random

async def cor(num):

n = random.randint(1, 10)

await asyncio.sleep(n)

return f'task {num} - sec. {n}'

result = asyncio.run(cor(5))

print(result)

--------------------

task 5 - sec. 4

Код вернул результат сопрограммы.

Функция **asyncio.run()** делает несколько важных вещей: создает новое событие, выполняет сопрограмму и возвращает результат, останавливает и завершает цикл событий. Она – главная точка входа в приложение asyncio.

1. Слово **await** приостанавливает сопрограмму на время выполнения длительной операции. Использование ключевого слова **await** приводит к выполнению сопрограммы, а не просто к возврату объекта сопрограммы, как при прямом вызове (пункт 1). Кроме того, выражение **await** приостанавливает объемлющую сопрограмму до того момента, как сопрограмма, которую мы ждем, завершится и вернет результат. А после этого мы получим доступ к возвращенному результату, а объемлющая сопрограмма пробудится и обработает результат.

import asyncio

import random

async def func(num):

n = random.randint(1, 10)

await asyncio.sleep(n)

return f'task {num} - sec. {n}'

async def main():

res\_a = await func(1)

res\_b = await func(2)

print(res\_a)

print(res\_b)

asyncio.run(main())

--------------------

task 1 - sec. 7

task 2 - sec. 7

--------------------

time: 0:00:14.013180

В примере выше две копии сопрограммы работают последовательно.

1. Чтобы выполнить код конкурентно нужны задачи. Задача – это обертка вокруг сопрограммы, которая планирует выполнение ее в цикле событий как можно раньше. И планирование, и выполнение происходят в *неблокирующем режиме*, т.е., создав задачу, можно сразу приступить к выполнению другого кода, пока эта задача работает в фоне.

Для создания задачи служит функция **asyncio.create\_task**(). Ей передается подлежащая выполнению сопрограмма, а в ответ она немедленно возвращает объект задачи. Этот объект можно включить в выражение **await**, которое извлечет возвращенное значение по завершении задачи.

Каждое обращение к **create\_task** возвращает управление немедленно!

import asyncio

import random

async def func(num):

n = random.randint(1, 10)

await asyncio.sleep(n)

return f'task {num} - sec. {n}'

async def main():

a = asyncio.create\_task(func(1))

b = asyncio.create\_task(func(2))

res\_a = await a

res\_b = await b

print(res\_a)

print(res\_b)

asyncio.run(main())

--------------------

task 1 - sec. 5

task 2 - sec. 3

--------------------

time: 0:00:05.012068

В примере выше две задачи на основе сопрограмм выполняются конкурентно. Сопрограмма main дойдя до внутренних сопрограмм приостанавливается и ждет выполнение всех задач. Внутренние сопрограммы выполняются конкурентно. Далее main возобнавляет работу. Задачи не работают конкурентно, конкурентно задачи только спят!

1. У каждого объекта задачи есть метод **cancel**, который можно вызвать, если требуется остановить задачу. В результате снятия задача возбудит исключение CancelledError, когда мы ждем ее с помощью await. Это исключения можно обработать.

import asyncio

from asyncio import CancelledError

import random

async def func(num):

n = random.randint(7, 10)

await asyncio.sleep(n)

return f'task {num} - sec. {n}'

async def main():

a = asyncio.create\_task(func(1))

cnt\_sec = 0

while not a.done():

await asyncio.sleep(1)

cnt\_sec += 1

print(cnt\_sec)

if cnt\_sec == 5:

a.cancel()

try:

res = await a

except CancelledError:

res = 'Stop by timeout > 5 secs.'

asyncio.run(main())

--------------------

Stop by timeout > 5 secs.

--------------------

time: 0:00:06.053051

В примере выше запускаем задачу, которая выполняется более 5 секунд, по прошествию 5 секунд останавливаем эту задачу. Вызов **cancel** прерывает задачу только если она уже находится в точке ожидания (await) или когда дойдет до следующей такой точки. В цикле while проверяем состояние задачи.

1. Автоматическая остановка задачи по таймауту с помощью функции **wait\_for().**

async def func(num):

n = random.randint(7, 10)

await asyncio.sleep(n)

return f'task {num} - sec. {n}'

async def main():

T = 2

a = asyncio.create\_task(func(1))

aw = asyncio.wait\_for(a, timeout=T)

try:

res = await aw

except TimeoutError:

res = f'Stop by timeout > {T} secs.'

asyncio.run(main())

--------------------

Stop by timeout > 2 secs.

--------------------

time: 0:00:02.008888

Функция **asyncio.shield()** предотвращает снятие задачи.

1. Объект **future** в Python содержит одно значение, которое мы ожидаем получить в будущем, но пока еще, возможно, не получили. Обычно, в момент создания, future не существует. Объект в таком состоянии называется неполным, неразрешенным или просто неготовым. И только получив результат, мы можем установить значение объекта future, в результате чего он становится полным и из него можно извлечь результат. Будущие объекты также можно использовать в выражениях await. Будущий объект будет находится в состоянии сна пока в нем не будет установлено значение.

from asyncio import Future

f = Future()

print(f.done())

f.set\_result(5)

print(f.done())

print(f.result())

---------------------

False

True

5

Создавая задачу, мы создаем пустой объект future и запускаем сопрограмму. А когда сопрограмма завершится с результатом или вследствие исключения, мы записываем этот результат или объект-исключение во future.

1. Цикл событий можно создать, воспользовавшись методом **asyncio.new\_event\_loop()**. Он возвращает экземпляр цикла событий, который дает доступ ко всем низкоуровневым методам, в частности методу **run\_until\_complete**, который принимает сопрограмму и исполняет ее до завершения. Закончив работу с циклом событий, мы должны закрыть его, чтобы освободить занятые ресурсы. Обычно это делается в блоке finally, чтобы цикл был закрыт даже в случае исключения.

import functools

import time

import random

import asyncio

def async\_timed():

def wrapper(func):

@functools.wraps(func)

async def wrapped(\*args, \*\*kwargs):

start = time.monotonic()

try:

return await func(\*args, \*\*kwargs)

finally:

end = time.monotonic()

total = end - start

print(f'{func.\_\_name\_\_}: {total:.4f} sec')

return wrapped

return wrapper

async def func(num):

n = random.randint(7, 10)

await asyncio.sleep(n)

return f'task {num}: {n} sec'

@async\_timed()

async def main():

a = asyncio.create\_task(func(1))

b = asyncio.create\_task(func(2))

print(await a)

print(await b)

loop = asyncio.new\_event\_loop()

try:

loop.run\_until\_complete(main())

finally:

loop.close()

------------------------

task 1: 8 sec

task 2: 3 sec

main: 8.0150 sec

Можно получить доступ к текущему запущенному циклу событий с помощью метода **asyncio.get\_running\_loop().**

1. Цикл событий asyncio позволяет прослушивать любой сигнал, указанный в методе **add\_sig-nal\_handler**. Функция принимает номер сигнала и функцию, которая должна вызываться при получении этого сигнала. Для демонстрации реализуем обработчик сигнала, который снимает все работающие задачи. В asyncio есть функция **asyncio.all\_tasks**, возвращающая множество всех работающих задач.

import asyncio

import random

import signal

from asyncio import CancelledError

def cancel\_tasks():

tasks = asyncio.all\_tasks()

print(f'start cancel tasks')

for task in tasks:

task.cancel()

async def func(n):

k = random.randint(5, 10)

await asyncio.sleep(k)

return f'task {n}: {k} sec'

async def main():

a = asyncio.create\_task(func(1))

b = asyncio.create\_task(func(2))

c = asyncio.create\_task(func(3))

loop.add\_signal\_handler(signal.SIGINT, cancel\_tasks)

print(await a)

print(await b)

print(await c)

loop = asyncio.new\_event\_loop()

try:

loop.run\_until\_complete(main())

except CancelledError:

print('cancelled exception in')

finally:

loop.close()

В примере выше происходит конкурентно запуск задач, пока они работают ловим сигнал прерывания приложения (Ctrl+C). Если сигнал пойман отрабатывается функция отмены всех задач. Далее возбуждается исключение CancelledError, которое перехватывается.

1. Функция **asyncio.gather**() принимает последовательность допускающих ожидание объектов и  запускает их конкурентно всего в одной строке кода. Если среди объектов есть сопрограмма, то она автоматически обертывает ее задачей, чтобы гарантировать конкурентное выполнение. **asyncio.gather** возвращает объект, допускающий ожидание. Если использовать его в выражении await, то выполнение будет приостановлено, пока не завершатся все переданные объекты. А когда это произойдет, asyncio.gather вернет список результатов работы.

import random

import asyncio

async def func(num):

n = random.randint(1, 5)

await asyncio.sleep(n)

return f'task {num}: {n} sec'

@async\_timed()

async def main():

funcs = []

for i in range(1, 10):

funcs.append(func(i))

res = await asyncio.gather(\*funcs)

print(res)

loop = asyncio.new\_event\_loop()

try:

loop.run\_until\_complete(main())

finally:

loop.close()

--------------------------------------

['task 1: 4 sec', 'task 2: 1 sec', 'task 3: 3 sec', 'task 4: 2 sec', 'task 5: 4 sec', 'task 6: 3 sec', 'task 7: 1 sec', 'task 8: 2 sec', 'task 9: 1 sec']

main: 4.0160 sec

Независимо от порядка завершения допускающих ожидание объектов, результаты гарантированно будут возвращены в том порядке, в каком объекты передавались.

1. asyncio.gather принимает необязательный параметр, **return\_exceptions**, который позволяет указать, как мы хотим обрабатывать исключения от допускающих ожидание объектов. Возможно два варианта:

return\_exceptions = False – режим по умолчанию. Если хотя бы одна сопрограмма возбуждает исключение, то gather возбуждает то же исключение в точке await. Но, даже если какая-то сопрограмма откажет, остальные не снимаются и продолжат работать при условии, что мы обработаем исключение и оно не приведет к остановке цикла событий и снятию задач;

return\_exceptions = True – в этом случае исключения возвращаются в том же списке, что результаты.

import asyncio

import random

async def func(num):

n = random.randint(1, 5)

await asyncio.sleep(n)

return f'task {num}: {n} sec'

async def func2(num):

raise Exception

@async\_timed()

async def main():

funcs = []

for i in range(1, 10):

if i%2:

funcs.append(func(i))

else:

funcs.append(func2(i))

res = await asyncio.gather(\*funcs, return\_exceptions=True)

print(res)

loop = asyncio.new\_event\_loop()

try:

loop.run\_until\_complete(main())

finally:

loop.close()

------------------------------------

['task 1: 5 sec', Exception(), 'task 3: 3 sec', Exception(), 'task 5: 5 sec', Exception(), 'task 7: 1 sec', Exception(), 'task 9: 1 sec']

main: 5.0000 sec

**asyncio.gather** не снимает другие работающие задачи из-за отказа. Во многих случаях это приемлемо, но вообще является недостатком. Второй недостаток – необходимость дождаться завершения всех сопрограмм, прежде чем можно будет приступить к обработке результатов.

1. Для решения проблемы которая заключается в ожидании завершения всех сопрограмм для получения результатов, в asyncio есть функция **as\_completed**(). Она принимает список допускающих ожидание объектов и возвращает итератор по будущим объектам. Эти объекты можно перебирать, применяя к каждому await. Когда выражение await вернет управление, мы получим результат первой завершившейся сопрограммы. Это значит, что мы сможем обрабатывать результаты по мере их доступности, но теперь порядок результатов не детерминирован, поскольку неизвестно, какой объект завершится первым.

async def func(num):

n = random.randint(1, 15)

await asyncio.sleep(n)

return f'task {num}: {n} sec'

@async\_timed()

async def main():

funcs = [func(i) for i in range(1, 10)]

for finished\_task in asyncio.as\_completed(funcs, timeout=5):

try:

print(await finished\_task)

except TimeoutError:

print('Timeout')

loop = asyncio.new\_event\_loop()

try:

loop.run\_until\_complete(main())

finally:

loop.close()

---------------------------------------

task 3: 1 sec

task 5: 2 sec

task 7: 3 sec

task 9: 5 sec

task 8: 6 sec

Timeout

Timeout

Timeout

Timeout

main: 7.0150 sec

Под капотом каждая сопрограмма обертывается задачей и начинает выполняться конкурентно. Функция немедленно возвращает итератор, который мы начинаем обходить. Войдя в цикл for, мы сразу натыкаемся на await finished\_task. Здесь выполнение приостанавливается до момента поступления первого результата. Затем снова дойдем до await finished\_task, и, так как запросы выполняются конкурентно, второй результат станет доступен, и т.д.

1. Функция **wait** в asyncio похожа на gather, но дает более точный контроль над ситуацией. У нее есть несколько параметров, позволяющих решить, когда мы хотим получить результаты. Кроме того, она возвращает два множества: задачи, завершившиеся успешно или в результате исключения, а также задачи, которые продолжают выполняться. Еще эта функция позволяет задать тайм-аут, который, не возбуждает исключений.

Базовая сигнатура wait – список допускающих ожидание объектов, за которым следует факультативный тайм-аут и факультативный параметр return\_when, который может принимать значения ALL\_COMPLETED, FIRST\_EXCEPTION и FIRST\_COMPLETED, а по умолчанию равен ALL\_COMPLETED.

ALL\_COMPLETED - в этом режиме функция ждет завершения всех задач и только потом возвращает управление.

async def func(num):

n = random.randint(1, 15)

await asyncio.sleep(n)

return f'task {num}: {n} sec'

@async\_timed()

async def main():

tasks = [asyncio.create\_task(func(i)) for i in range(1, 10)]

done, pending = await asyncio.wait(tasks, return\_when=asyncio.ALL\_COMPLETED)

for d in done:

print(await d)

loop = asyncio.new\_event\_loop()

try:

loop.run\_until\_complete(main())

finally:

loop.close()

----------------------------------------

task 3: 4 sec

task 9: 5 sec

task 6: 13 sec

task 1: 10 sec

task 8: 4 sec

task 4: 11 sec

task 7: 1 sec

task 2: 6 sec

task 5: 15 sec

main: 15.0160 sec

Предложение await wait вернет управление, когда все запросы завершатся, и мы получим два множества: завершившиеся задачи и еще работающие задачи. Множество done содержит все задачи, которые завершились успешно или в результате исключения, а множество pending – еще не завершившиеся задачи. В данном случае мы задали режим ALL\_COMPLETED, поэтому множество pending будет пустым, так как asyncio.wait не вернется, пока все не завершится. При возникновении исключения, есть несколько способов обработать его. Можно выполнить await и дать возможность исключению распространиться выше, можно выполнить await к выполненой задаче, обернуть его в блок try/except, чтобы обработать исключение, или воспользоваться методами task.result() и task.exception().

FIRST\_EXCEPTION -1) Если ни в одной задаче не было исключений, то этот режим эквивалентен ALL\_COMPLETED. 2) Если хотя бы в одной задаче возникло исключение, то wait немедленно возвращается. Множество done будет содержать как задачи, завершивши-

еся успешно, так и те, в которых имело место исключение. Множество pending может быть пустым, а может содержать задачи, которые продолжают выполняться.

async def func(num):

n = random.randint(1, 15)

await asyncio.sleep(n)

return f'task {num}: {n} sec'

async def func2(num):

n = random.randint(7, 10)

await asyncio.sleep(n)

raise TimeoutError

@async\_timed()

async def main():

tasks = []

for i in range(1, 10):

if i % 2:

f = func

else:

f = func2

task = asyncio.create\_task(f(i))

task.set\_name(f'task {i}')

tasks.append(task)

done, pending = await asyncio.wait(tasks, return\_when=asyncio.FIRST\_EXCEPTION)

for d in done:

if d.exception() is None:

print(await d)

else:

print('Exception')

for p in pending:

print(f'pending ... {p.get\_name()}')

loop = asyncio.new\_event\_loop()

try:

loop.run\_until\_complete(main())

finally:

loop.close()

------------------------------------------

task 9: 6 sec

task 7: 6 sec

task 1: 4 sec

Exception

pending ... task 3

pending ... task 8

pending ... task 4

pending ... task 2

pending ... task 5

main: 7.0310 sec

FIRST\_COMPLETED. В этом режиме wait возвращает управление, как только получен хотя бы один результат. Это может быть, как успешно завершившаяся задача, так и задача, в которой возникло исключение. Остальные задачи можно либо снять, либо дать им возможность продолжать работу.

...

async def func(num):

n = random.randint(1, 15)

await asyncio.sleep(n)

return f'task {num}: {n} sec [done]'

async def func2(num):

n = random.randint(7, 10)

await asyncio.sleep(n)

raise TimeoutError

@async\_timed()

async def main():

pending = []

for i in range(1, 10):

if i % 2:

f = func

else:

f = func2

task = asyncio.create\_task(f(i))

task.set\_name(f'task {i}')

pending.append(task)

while pending:

done, pending = await asyncio.wait(pending, return\_when=asyncio.FIRST\_COMPLETED)

for d in done:

if d.exception() is None:

print(await d)

else:

d.cancel()

print(f'{d.get\_name()}: [cancel]')

print(f'pending tasks: {len(pending)}')

loop = asyncio.new\_event\_loop()

try:

loop.run\_until\_complete(main())

finally:

loop.close()

-----------------------------------

task 3: 6 sec [done]

pending tasks: 8

task 8: [cancel]

task 4: [cancel]

task 2: [cancel]

pending tasks: 5

task 6: [cancel]

pending tasks: 4

task 1: 10 sec [done]

pending tasks: 3

task 7: 11 sec [done]

pending tasks: 2

task 9: 12 sec [done]

pending tasks: 1

task 5: 13 sec [done]

pending tasks: 0

main: 13.0150 sec

1. Multiprocessing

import functools

import time

from multiprocessing import Process

def timed(func):

@functools.wraps(func)

def wrap(\*args):

start = time.monotonic()

res = func(\*args)

total = time.monotonic() - start

print(f'{func.\_\_name\_\_}: {total:.4f} sec')

return res

return wrap

@timed

def calc\_func(n):

s = 0

for i in range(n):

s += i\*\*n

return s

@timed

def main():

a = Process(target=calc\_func, args=(6000,))

b = Process(target=calc\_func, args=(7000,))

a.start()

b.start()

a.join()

b.join()

if \_\_name\_\_ == "\_\_main\_\_":

main()

------------------------------

calc\_func: 2.8590 sec

calc\_func: 4.3430 sec

main: 4.5310 sec

Параллельное выполнение двух вычислительных функций. Два процесса. Этот API годится для простых случаев, но, очевидно, не работает, если нужно получать возвращенное функцией значение или обрабатывать результаты по мере готовности.

1. Для счетных задач чтобы получить значение функций и выполнить эти функции параллельно, необходимо использовать пул процессов.

При создании пула автоматически создается столько процессов, сколько имеется процессорных ядер на данной машине. Количество ядер можно получить от функции multiprocessing.cpu\_count(). Если это не годится, то можно передать функции Pool() произвольное целое число в аргументе processes.

def calc\_func(n, j):

start = time.monotonic()

s = 0

for i in range(n):

s += i\*\*n

total = time.monotonic() - start

return f'func {j}: {total:.4f} sec [done]'

@timed

def main(k):

proc = []

with Pool() as pool:

for i in range(1, k+1):

t = random.randint(3000, 8000)

proc.append(pool.apply\_async(calc\_func, args=(t, i)))

for v in proc:

print(v.get())

if \_\_name\_\_ == "\_\_main\_\_":

main(10)

--------------------------------------

func 1: 3.0790 sec [done]

func 2: 0.6250 sec [done]

func 3: 3.3130 sec [done]

func 4: 3.5620 sec [done]

func 5: 0.9690 sec [done]

func 6: 3.9220 sec [done]

func 7: 1.2500 sec [done]

func 8: 4.9530 sec [done]

func 9: 1.4380 sec [done]

func 10: 0.9370 sec [done]

main: 8.5630 sec