Работа с короутинами и asyncio

Task

В модуле про event loop мы создавали фоновую программу-демона, которая работает всегда.

Используя метод **create_task**, мы создавали **asyncio.Task** из нашей короутины и отправляли в цикл событий:

```
...
loop = asyncio.get_event_loop()
loop.create_task(worker())
loop.run_forever()
```

Внутри короутин мы можем использовать аналогичный метод, но **loop** уже не нужен. Библиотека **asyncio** все сделает за нас.

В примере ниже мы запускаем аналогичного демона, но инициализируем воркеры не напрямую с использованием **event loop**, а из другой короутины:

```
import asyncio


async def worker(idx: int):
    while True:
        print(f"worker-{idx} msg")
        await asyncio.sleep(1)


async def main():
    for i in range(3): # добавляем в event loop 3 task-и
        asyncio.create_task(worker(i + 1))

    while True:
        print(f"main msg")
        await asyncio.sleep(1) # сон нужен для передачи управления другим короутинам, нашим worker-ам


asyncio.run(main())
```

```
main msg
worker-1 msg
worker-2 msg
worker-3 msg
main msg
worker-1 msg
worker-2 msg
worker-3 msg
```

Awaitable

В python есть понятие **awaitable**.

Это все объекты, с которыми мы можем использовать ключевое слово **await**. Среди них:

- Coroutines
- Tasks
- Futures

С сущностью **Future** мы познакомимся позднее в курсе более плотно. А пока нам достаточно тезиса, что **asyncio.Future** — это более низкоуровневое представление **asyncio.Task**

```
import asyncio


async def coro():
    await asyncio.sleep(5)


async def main():
    await coro() # дожидаемся выполнения короутины

    task = asyncio.create_task(coro()) # создаем Task и добавляем ее в event loop

    await task # дожидаемся выполнения Task-и


asyncio.run(main())
```

Запуск нескольких короутин

Нам нужно одновременно запустить несколько короутин и продолжить выполнение программы после завершения их работы. Для этого отлично подойдет **`asyncio.gather`:**

```
import asyncio
import random


async def random_sleep(i: int):
    delay = random.randint(1, 5)
    print(f"{i}: start sleep")
    await asyncio.sleep(delay)
    print(f"{i}: end sleep")


async def main():
    await asyncio.gather(
        random_sleep(1),
        random_sleep(2),
        random_sleep(3),
        random_sleep(4),
        random_sleep(5),
    )


asyncio.run(main())
```

```
1: start sleep
2: start sleep
3: start sleep
4: start sleep
5: start sleep
2: end sleep
3: end sleep
4: end sleep
1: end sleep
5: end sleep
```

Подобную логику можно реализовать с помощью другой функции — **`asyncio.wait`**. **return_when=ALL_COMPLETED** в этом примере не обязательно, это значение используется при вызове wait по умолчанию.

```
import asyncio
import random
from asyncio import ALL_COMPLETED


async def random_sleep(i: int):
    delay = random.randint(1, 5)
    print(f"{i}: start sleep")
    await asyncio.sleep(delay)
    print(f"{i}: end sleep")


async def main():
    await asyncio.wait([random_sleep(i) for i in range(1, 6)], return_when=ALL_COMPLETED)


asyncio.run(main())
```

Теперь представим, что у нас есть несколько одинаковых источников, где мы можем получить интересующую нас информацию. Главное для нас — выигрыш в скорости.

Другими словами, мы хотим получить результат того запроса, который отработает быстрее. Используя ранее упомянутую **`asyncio.wait`**, обработаем и этот случай, указав параметр **return_when=asyncio.FIRST_COMPLETED**.

В этом случае, как только любая **Task** завершит работу, функция **wait** вернет управление.

```
import asyncio
import random
from asyncio import FIRST_COMPLETED


async def random_sleep(i: int):
    delay = random.randint(1, 5)
    print(f"{i}: start sleep")
    await asyncio.sleep(delay)
    print(f"{i}: end sleep")


async def main():
    await asyncio.wait([random_sleep(i) for i in range(1, 6)], return_when=FIRST_COMPLETED)


asyncio.run(main())
```

```
5: start sleep
1: start sleep
3: start sleep
4: start sleep
2: start sleep
4: end sleep
```

Возвращение управления по timeout

При разработке нашего сервера мы делаем различные запросы по сети. Представим, что метод нашего API запрашивает какие-то опциональные данные, без которых ответ все еще будет валидным.

Защитим себя от проблем сторонних сервисов. Мы не хотим возвращать с сервера 500-й код ответа из-за проблем сторонних сервисов. Для этого воспользуемся функциями **`wait_for`** или **`wait`** и передадим значение таймаута. По истечению заданного интервала времени программа вернет управление, и мы сможем обработать запрос.

Пример. У нас запросили карту города. Это основная информация, но при возможности мы дополнительно передаем информацию о дорожных пробках.

```
import asyncio


async def request_traffic_jams():
    await asyncio.sleep(10)


async def main():
    try:
        await asyncio.wait_for(request_traffic_jams(), timeout=1.0)
    except asyncio.TimeoutError:
        print("Problems with traffic_jams service")


asyncio.run(main())
```

```
Problems with traffic_jams service
```

Корректное завершение программы

Нам нужно перезапустить сервер. Он работает под нагрузкой, и в каждый момент времени происходят операции записи данных в базу. Мы не можем просто убить процесс, потому что в этом случае мы потеряем какие-то важные данные. Для этого нужно безопасно завершить программу, дождавшись всех походов в базу.

Можно обработать исключение **`CancelledError`**. Отменяем все task и дожидаемся тех, что не завершились после отмены.

```
import asyncio
from asyncio import CancelledError


async def db_operation():
    print("operation start")
    try:
        while True:
            print("operation new round")
            await asyncio.sleep(1)
    except CancelledError:
        print("operation was cancelled")
        await asyncio.sleep(1)

    print("operation done")


async def main():
    task = asyncio.create_task(db_operation())
    await asyncio.sleep(5)

    task.cancel() # вызываем исключение CancelledError внутри db_operation, оно будет обработано как только
event loop передаст управление task-е, обрабатывающей эту короутину

    await task # передаем управление db_operation для корректного завершения


asyncio.run(main())
```

```
operation start
operation new round
operation new round
operation new round
operation new round
operation new round
operation was cancelled
operation done
```

Приведем общий пример. У нас работает несколько worker-ов. В случае если в 1 из них было вызвано исключение - хотим корректно завершить программу - отменить другие asyncio.Task-и. Функция **asyncio.wait** с параметром **return_when=asyncio.FIRST_EXCEPTION** вернет управление в **main()**. Внутри переменной pending будут перечислены все taks-и, которые еще обрабатываются в данный момент.

```
import asyncio
from asyncio import CancelledError


async def worker(idx: int):
    try:
        while True:
            print(f"worker {idx} new iter")
            await asyncio.sleep(1)

    except CancelledError:
        print(f"worker was cancelled")

    print(f"worker was correctly cancelled")


async def worker_with_exception(idx: int):
    await asyncio.sleep(1)
    print(f"worker {idx} Exception")
    raise Exception


async def main():
    done, pending = await asyncio.wait(
        [*[worker(i) for i in range(1, 4)], worker_with_exception(4)],
        return_when=asyncio.FIRST_EXCEPTION,
    )

    print(f"done({len(done)}) pending({len(pending)})") # показываем число завершенный и работающих task-ов

    for p in pending:
        p.cancel()

    await asyncio.gather(*pending)


asyncio.run(main())
```

```
worker 1 new iter
worker 2 new iter
worker 3 new iter
worker 4 Exception
worker 1 new iter
worker 2 new iter
worker 3 new iter
done(1) pending(3)
worker was cancelled
worker was correctly cancelled
worker was cancelled
worker was correctly cancelled
worker was cancelled
worker was correctly cancelled
```