Asyncio

- Сопрограммы
- Типы future и awaitable
- Асинхронный контекстный менеджер
- Aiohttp

??????????

ЭТО ТЕХНОЛОГИЯ УСКОРЕНИЯ РАБОТЫ В ОДНОМ ПОТОКЕ. ДЛЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ THREADING,

Сопрограммы - функции с возможностью приостановки при длительной внешней операции. Async определяет сопрограмму, await приостанавливает сопрограмму на время выполнения внешней задачи. При вызове сопрограммы она напрямую не выполняется. Для старта нужна точка входа в асинхронные вычисления.

Пример: правильное использование.

```
import asyncio


async def corutine_add_one(a: int) -> int:
    return b + 1


corutine_res = asyncio.run(corutine_add_one(5))
print(type(corutine_res), '  ', corutine_res)
```

Однако если просто запустить корутину, то результат будет интересным:

```
import asyncio


async def corutine_add_one(a: int) -> int:
    return b + 1


corutine_res = corutine_add_one(5)
print(type(corutine_res), '  ', corutine_res)
```

Вывод:

```
<class 'coroutine'>    <coroutine object corutine_add_one at 0x0000026850FD4270>
```

Существует цикл событий. При постановке корутины на паузу задача передается следующей корутине. Однако такой код выполняется в виде простого синхронного кода:

```
import asyncio

async def sleep_10(x: int) -> int:
    await asyncio.sleep(10)
    return x+10

async def main():
    x1 = await sleep_10(1)
    x2 = await sleep_10(1)
    print(x1, '  ', x2)

asyncio.run(main())
```

Задачи

Для приближения к параллельным вычислениям необходимо использовать задачи.

```
import asyncio
from util import delay

async def main():
    sleep1 = asyncio.create_task(delay(10))
    sleep2 = asyncio.create_task(delay(10))
    await sleep1
    await sleep2

asyncio.run(main())
```

К тому же, если убрать await sleep2, то вторая задача все равно выполнится, т.к. вроде все задачи из очереди запускаются. Это так, вот только не говорится о том, завершаются они или нет. Вот пример кода:

```
async def delay(delay_seconds: int) -> int:
    print(f'Засыпаю на {delay_seconds} секунд.')
    await asyncio.sleep(delay_seconds)
    print(f'сон в течение {delay_seconds} закончился.')
    return delay_seconds

async def delay_with_inner(delay_seconds: int) -> int:
    print(f'Засыпаю на {delay_seconds} секунд.')
    t1 = asyncio.create_task(delay(15))
```

```
    await asyncio.sleep(delay_seconds)
    #await t1
    print(f'сон в течение {delay_seconds} закончился.')
    return delay_seconds

async def main():
    sleep1 = asyncio.create_task(delay_with_inner(10))
    sleep2 = asyncio.create_task(delay_with_inner(10))
    sleep3 = asyncio.create_task(delay_with_inner(10))
    await sleep1
    await sleep2
    await sleep3

asyncio.run(main())
```

Если внешнюю мы запускаем на 10 секунд, то внутренняя (t1) тоже запускается - т.к. находится в очереди задач. Но вот только программа завершается до завершения t1.

```
Засыпаю на 10 секунд.
Засыпаю на 10 секунд.
Засыпаю на 10 секунд.
Засыпаю на 15 секунд.
Засыпаю на 15 секунд.
Засыпаю на 15 секунд.
сон в течение 10 закончился.
сон в течение 10 закончился.
сон в течение 10 закончился.
```

И все! А если раскомментировать строку с ожиданием t1, то тогда сначала завершится t1

```
Засыпаю на 10 секунд.
Засыпаю на 10 секунд.
Засыпаю на 10 секунд.
Засыпаю на 15 секунд.
Засыпаю на 15 секунд.
Засыпаю на 15 секунд.
сон в течение 15 закончился.
сон в течение 15 закончился.
сон в течение 15 закончился.
сон в течение 10 закончился.
```

```
сон в течение 10 закончился.
сон в течение 10 закончился.
```

А для такого кода будет создано 3 файла. Но если вместо sleep3 установить sleep1, то все задачи стартуют, но будет создан только файл 5.txt.

```
import asyncio
from util import delay_with_inner, delay, delay_and_writefile
from time import sleep

async def main():
    sleep1 = asyncio.create_task(delay_and_writefile(5))
    sleep2 = asyncio.create_task(delay_and_writefile(10))
    sleep3 = asyncio.create_task(delay_and_writefile(11))
    await sleep3

asyncio.run(main())
```

Исходя из этого, важна не последовательность помещения в задачи, а последовательность при обращении (момент await).

Снятие задач и тайм-ауты

```
import asyncio
from asyncio import CancelledError
from util import delay
async def main():
    long_task = asyncio.create_task(delay(10))
    seconds_elapsed = 0
    while not long_task.done():
        print('Задача не закончилась, следующая проверка через секунду.')
        await asyncio.sleep(1)
        seconds_elapsed = seconds_elapsed + 1
        if seconds_elapsed == 5:
            long_task.cancel()
    try:
        await long_task
    except CancelledError:
        print('Наша задача была снята')
asyncio.run(main())
```

Тайм-аут:

```
import asyncio
from util import delay
async def main():
    delay_task = asyncio.create_task(delay(2))
    try:
        result = await asyncio.wait_for(delay_task, timeout=1)
        print(result)
    except asyncio.exceptions.TimeoutError:
        print('Тайм-аут!')
        print(f'Задача была снята? {delay_task.cancelled()}')
asyncio.run(main())
```

При помощи shield можно игнорировать запросы на снятие:

```
import asyncio
from util import delay
async def main():
    task = asyncio.create_task(delay(10))
    try:
    result = await asyncio.wait_for(asyncio.shield(task), 5)
        print(result)
    except TimeoutError:
        print("Задача заняла более 5 с, скоро она закончится!")
        result = await task
        print(result)
asyncio.run(main())
```

Список задач:

```
for task in asyncio.tasks.all_tasks():
    print(task)
```

Множественные task в цикле.

Проблема: при создании в цикле при возникновении исключения в одной задаче, остальные задачи могут не завершить выполнение.

Функция gather

```
import asyncio
import aiohttp
from aiohttp import ClientSession
from chapter_04 import fetch_status

async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https://example.com' for _ in range(1000)]
        requests = [fetch_status(session, url) for url in urls]
        status_codes = await asyncio.gather(*requests)
        print(status_codes)
asyncio.run(main())
```

Gather возвращает статусы в порядке передаче объектов, независимо от времени исполнения. Возвращает все данные одновременно.

Обработка исключений - необязательный bool параметр return_exceptions.

- return_exceptions=False 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. Сам по себе вызов gather не возбуждает исключений, и мы можем обработать исключения, как нам удобно.

```
async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https://example.com', 'python://example.com']
        tasks = [fetch_status_code(session, url) for url in urls]
        results = await asyncio.gather(*tasks, return_exceptions=True)
        exceptions = [res for res in results if isinstance(res, Exception)]
        successful_results = [res for res in results if not isinstance(res, Exception)]
        print(f'Все результаты: {results}')
        print(f'Завершились успешно: {successful_results}')
        print(f'Завершились с исключением: {exceptions}')
```

Функция as_completed

Начинает возвращать данные по мере поступления. Однако порядок не определен.

```
import asyncio
import aiohttp
from aiohttp import ClientSession
from chapter_04 import fetch_status

async def main():
    async with aiohttp.ClientSession() as session:
        fetchers = [fetch_status(session, 'https://www.example.com', 1),
                    fetch_status(session, 'https://www.example.com', 1),
                    fetch_status(session, 'https://www.example.com', 10)]
        for finished_task in asyncio.as_completed(fetchers):
            print(await finished_task)
asyncio.run(main())
```

asyncio.as_completed(fetchers, timeout=2) задает тайм-аут. Но созданные задачи продолжают работать в фоновом режиме.

Точный контроль над функциями

Используется wait:

```
async def main():
    async with aiohttp.ClientSession() as session:
    fetchers = \
        [asyncio.create_task(fetch_status(session, 'https://example.com')),
        asyncio.create_task(fetch_status(session, 'https://example.com'))]
    done, pending = await asyncio.wait(fetchers)
    print(f'Число завершившихся задач: {len(done)}')
    print(f'Число ожидающих задач: {len(pending)}')
    for done_task in done:
        result = await done_task
        print(result)
```

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.

wait не возбуждает исключений, исключения содержатся в методе done_task.exception()

Пример обработки исключений

```
import asyncio
import logging

async def main():
    async with aiohttp.ClientSession() as session:
        good_request = fetch_status(session, 'https://www.example.com')
        bad_request = fetch_status(session, 'python://bad')
        fetchers = [asyncio.create_task(good_request),
                    asyncio.create_task(bad_request)]
        done, pending = await asyncio.wait(fetchers)
        print(f'Число завершившихся задач: {len(done)}')
        print(f'Число ожидающих задач: {len(pending)}')
        for done_task in done:
            # result = await done_task возбудит исключение
            if done_task.exception() is None:
                print(done_task.result())
            else:
                logging.error("При выполнении запроса возникло исключение",
                              exc_info=done_task.exception())
asyncio.run(main())
```

???? future ? awaitable

Практически редко применяются, но нужны для понимания

Будущие объекты можно использовать в выражениях await. Это означает «я посплю, пока в будущем объекте не будет установлено значение, с которым я могу работать, а когда оно появится, разбуди меня и дай возможность его обработать».

```
from asyncio import Future
my_future = Future()
print(f'my_future готов? {my_future.done()}')
my_future.set_result(42)
print(f'my_future готов? {my_future.done()}')
print(f'Какой результат хранится в my_future? {my_future.result()}')
```

Вариант для использования не до конца определенной переменной:

```
from asyncio import Future
import asyncio
def make_request() -> Future:
    future = Future()
    asyncio.create_task(set_future_value(future))
    return future
async def set_future_value(future) -> None:
    await asyncio.sleep(1)
    future.set_result(42)
async def main():
    future = make_request()
    print(f'Будущий объект готов? {future.done()}')
    value = await future
    print(f'Будущий объект готов? {future.done()}')
    print(value)
asyncio.run(main())
```

Любой объект, который реализует метод __await__, можно использовать в выражении await. Это объекты, допускающие ожидание (Awaitaible).

??????????? ??????????? ????????

Асинхронный контекстный менеджер.

Это класс, реализующий два специальных метода-сопрограммы: __aenter__, который асинхронно захватывает ресурс, и __aexit__, который закрывает ресурс. Сопрограмма __aexit__ принимает несколько аргументов, относящихся к обработке исключений. Пример для сокетов:

```
import asyncio
import socket
from types import TracebackType
from typing import Optional, Type
class ConnectedSocket:
    def __init__(self, server_socket):
        self._connection = None
        self._server_socket = server_socket
    async def __aenter__(self):
        print('Вход в контекстный менеджер, ожидание подключения')
        loop = asyncio.get_event_loop()
        connection, address = await loop.sock_accept(self._server_socket)
        self._connection = connection
        print('Подключение подтверждено')
        return self._connection
    async def __aexit__(self,
                        exc_type: Optional[Type[BaseException]],
                        exc_val: Optional[BaseException],
                        exc_tb: Optional[TracebackType]):
        print('Выход из контекстного менеджера')
        self._connection.close()
        print('Подключение закрыто')
async def main():
    loop = asyncio.get_event_loop()
    server_socket = socket.socket()
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server_address = ('127.0.0.1', 8000)
```

```
    server_socket.setblocking(False)
    server_socket.bind(server_address)
    server_socket.listen()
    async with ConnectedSocket(server_socket) as connection:
        data = await loop.sock_recv(connection, 1024)
        print(data)
asyncio.run(main())
```

Aiohttp

Сеансовый асинхронный http(s) клиент с автоматической поддержкой cookies. Пул подключений использует один сеанс.

Установка:

```
pip install aiohttp
```

Использование:

```
import asyncio
import aiohttp
from aiohttp import ClientSession
from util import async_timed

async def fetch_status(session: ClientSession, url: str) -> int:
    async with session.get(url) as result:
        return result.status

async def main():
    async with aiohttp.ClientSession() as session:
        url = 'https://www.example.com'
        status = await fetch_status(session, url)
        print(f'Состояние для {url} было равно {status}')
asyncio.run(main())
```

По умолчанию в сеансе не более 100 подключений. Для увеличения можно создать экземпляр класса TCPConnector, входящего в состав aiohttp, указав максимальное число подключений, и передать его конструктору ClientSession. Подробнее в документации aiohttp.

Тайм-аут:

По умолчанию тайм-аут запроса 5 минут. Можно устанавливать на уровне сеанса или запроса.

```
import asyncio
import aiohttp
from aiohttp import ClientSession
async def fetch_status(session: ClientSession, url: str) -> int:
```

```
    ten_millis = aiohttp.ClientTimeout(total=.01)
    async with session.get(url, timeout=ten_millis) as result:
        return result.status

async def main():
    session_timeout = aiohttp.ClientTimeout(total=1, connect=.1)
    async with aiohttp.ClientSession(timeout=session_timeout) as session:
        await fetch_status(session, 'https://example.com')
asyncio.run(main())
```

В этом случае полный тайм-аут 1 секунда, для установки соединения - 100мс. В функции fetch_status переопределяется в 10мс.

Множественные запросы

```
import asyncio
import aiohttp
from aiohttp import ClientSession
from chapter_04 import fetch_status

async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https://example.com' for _ in range(1000)]
        requests = [fetch_status(session, url) for url in urls]
        status_codes = await asyncio.gather(*requests)
        print(status_codes)
asyncio.run(main())
```

Обработка ошибок

```
async def main():
    async with aiohttp.ClientSession() as session:
        urls = ['https://example.com', 'python://example.com']
        tasks = [fetch_status_code(session, url) for url in urls]
        results = await asyncio.gather(*tasks, return_exceptions=True)
        exceptions = [res for res in results if isinstance(res, Exception)]
        successful_results = [res for res in results if not isinstance(res, Exception)]
        print(f'Все результаты: {results}')
        print(f'Завершились успешно: {successful_results}')
        print(f'Завершились с исключением: {exceptions}')
```