



Backend разработка на Python

Лекция 11

Асинхронное программирование

Кандауров Геннадий



Расскажите ваши впечатления о курсе

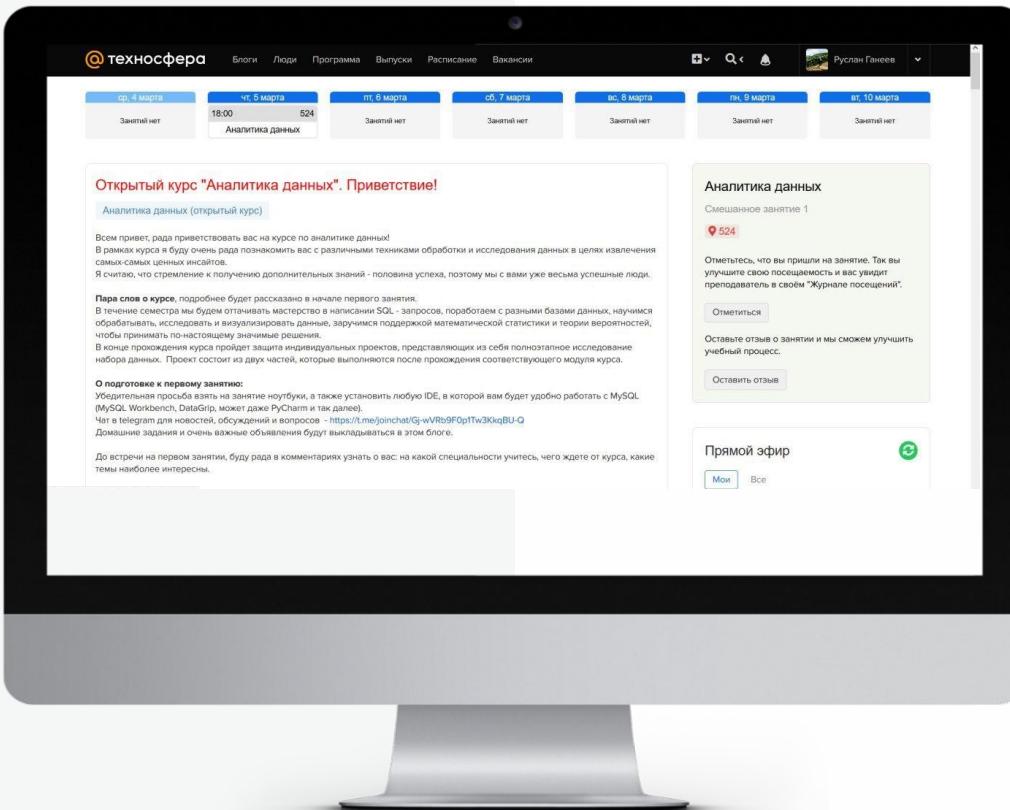
Мы в Отделе образовательных проектов Mail.ru Group проводим исследование качества наших курсов.

Ваша обратная связь очень важна для нас: так мы сможем делать наши проекты для студентов лучше.

Просим пройти опрос, заполнение анкеты займет не более 7 минут.

Как это сделать?

1. Проверьте свою почту, а также папку «спам».
2. Найдите письмо с темой «Оцените обучение на курсе...»
3. Пройдите опрос
4. 😊 Profit 😊



Напоминание отметиться на портале

+ отзывы после лекции

Содержание занятия

- Асинхронное программирование
- Event loop
- Корутины, нативные корутины
- asyncio
- Web фреймворки

IO bound vs CPU bound



Блокирующие операции

```
import socket

server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_sock.bind(('localhost', 15000))
server_sock.listen()

while True:
    client_sock, addr = server_sock.accept()

    while True:
        data = client_sock.recv(4096)
        if not data:
            break
        else:
            client_sock.send(data.decode().upper().encode())
    client_sock.close()
```

Блокирующие операции

- connect, accept, recv, send - блокирующие операции
- C10k problem, <http://kegel.com/c10k.html>
- Потоки дорого стоят (CPU & RAM)
- Потоки простаивают часть времени

Неблокирующие операции

Системные вызовы:

- select (man 2 select)
- poll (man 2 poll)
- epoll (man 7 epoll)
- kqueue

python:

- select
- selectors

select

```
def event_loop():
    while True:
        ready_to_read, _, _ = select(to_monitor, [], [])

        for sock in ready_to_read:
            if sock is server_sock:
                accept_conn(sock)
            else:
                respond(sock)
```

selectors

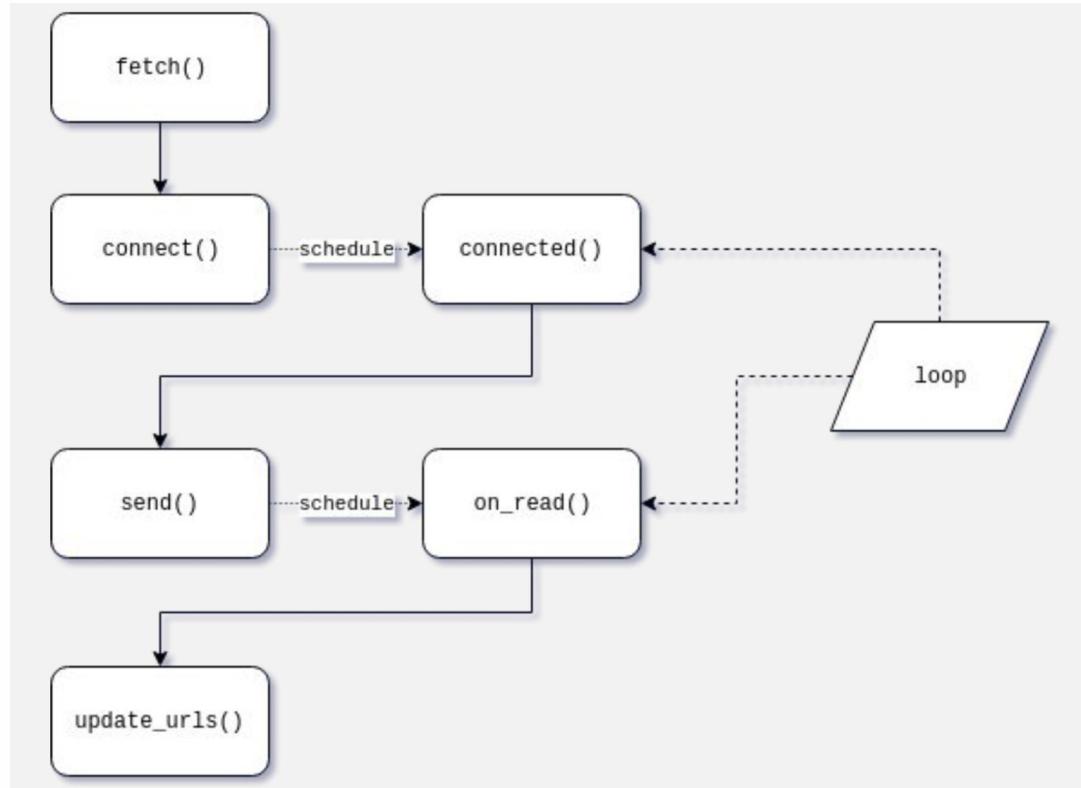
```
import selectors

selector = selectors.DefaultSelector()
selector.register(server_sock, selectors.EVENT_READ, accept_conn)

def event_loop():
    while True:
        events = selector.select() # (key, events_mask)

        for key, _ in events:
            # key: NamedTuple(fileobj, events, data)
            callback = key.data
            callback(key.fileobj)
            # selector.unregister(key.fileobj)
```

Callback hell



generator based event loop

Дэвид Бизли (David Beazley), "Python Concurrency From the Ground Up: LIVE!"

```
def event_loop():
    while any([tasks, to_read, to_write]):
        while not tasks:
            ready_to_read, ready_to_write, _ = select(to_read, to_write, [])
            for sock in ready_to_read:
                tasks.append(to_read.pop(sock))
            for sock in ready_to_write:
                tasks.append(to_write.pop(sock))
        try:
            task = tasks.pop(0)
            op_type, sock = next(task)
            if op_type == 'read':
                to_read[sock] = task
            elif op_type == 'write':
                to_write[sock] = task
        except StopIteration:
            pass
```

Корутины

```
def grep(pattern):
    print('start grep for', pattern)
    while True:
        s = yield
        if pattern in s:
            print('found!', s)
        else:
            print('no %s in %s' % (pattern, s))

g = grep('python')
next(g)
g.send('data')
g.send('deep python')

$ python generator_socket.py
start grep for python
no python in data
found! deep python
```

Корутины

- использование ***yield*** более обобщенно определяет корутину
- не только генерируют значения
- потребляют данные, отправленные через ***.send***
- отправленные данные возвращаются через ***data = yield***

Нативные корутины

Coroutine

Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the `async def` statement. See also [PEP 492](#).

Нативные корутины

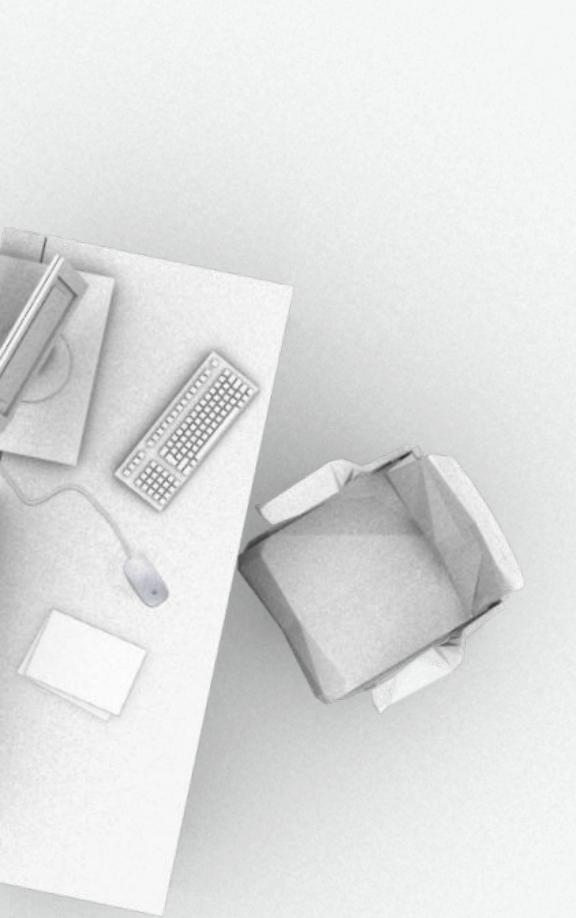
```
import asyncio, time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")
    await say_after(1, 'hello')
    await say_after(2, 'world')
    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())

>run.py
started at 16:42:46
hello
world
finished at 16:42:49
```



asyncio

asyncio

- 1 процесс
- 1 поток
- кооперативная многозадачность (vs вытесняющая)
- передача управления в event loop на ожидающих операциях
- async/await это API Python, а не часть asyncio

asyncio

Event loop:

coroutine > Task (Future)

- **Future** представляет ожидаемый в будущем (eventual) результат асинхронной операции;
- **Task** это *Future-like* объект, запускающий корутины в событийном цикле;
- **Task** используется для запуска нескольких корутин в событийном цикле параллельно.

asyncio

High-level APIs

- Coroutines and Tasks
- Streams
- Synchronization Primitives
- Subprocesses
- Queues
- Exceptions

asyncio

Low-level APIs

- Event Loop
- Futures
- Transports and Protocols
- Policies
- Platform Support

asyncio

Вспомогательное API

- `asyncio.create_task`
- `asyncio.sleep`
- `asyncio.gather`
- `asyncio.shield`
- `asyncio.wait_for`
- `asyncio.wait`
- `asyncio.Queue`
- `asyncio.Lock`
- `asyncio.Event`

Асинхронные фреймворки

- aiohttp <https://docs.aiohttp.org/en/stable/>
- tornado <https://www.tornadoweb.org/en/stable/>
- sanic <https://sanic.readthedocs.io/en/latest/>
- django (частично) <https://www.djangoproject.com/>

Домашнее задание по лекции 11

ДЗ #11

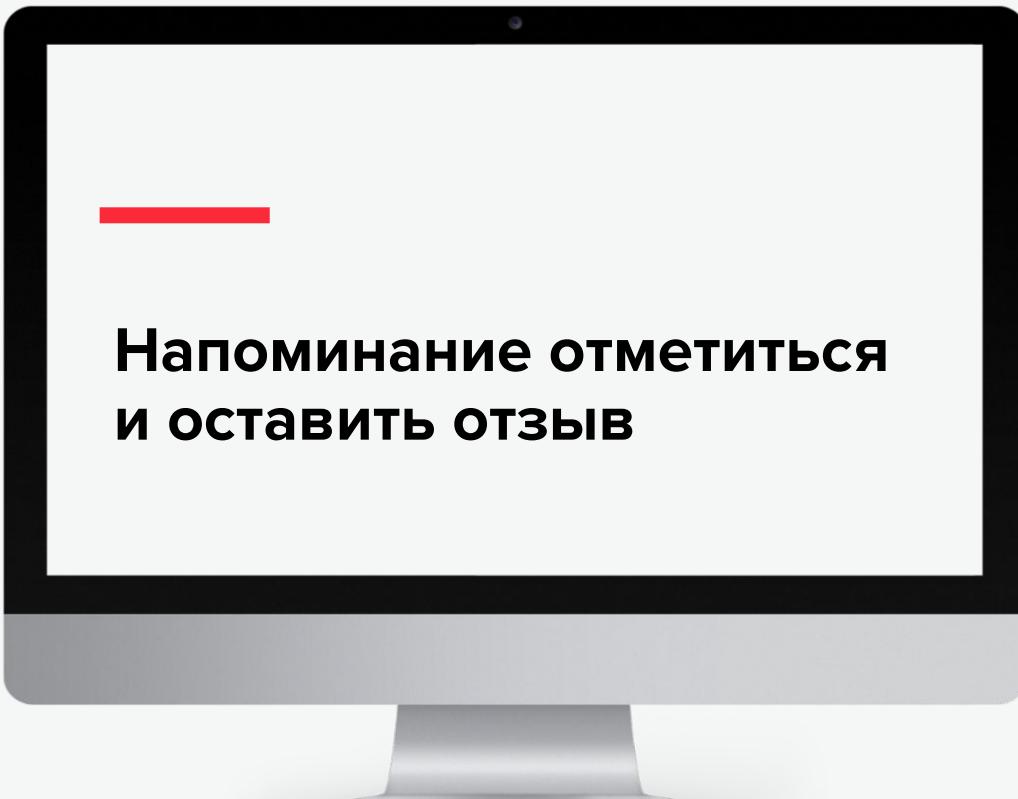
9

29.12.2020

баллов за
задание

срок сдачи

- Написать скрипт для обкачки списка урлов с возможностью задавать количество одновременных запросов, используя асинхронное программирование. Клиент можно использовать любой, например, из aiohttp. Так, 10 одновременных запросов могут задаваться командой:
`python fetcher.py -c 10 urls.txt`
- Асинхронная запись файлов на диск



**Напоминание отметить
и оставить отзыв**



**СПАСИБО
ЗА ВНИМАНИЕ**

