Обработка запросов в один поток

- Подходы для организации выполнения кода в одном потоке
- Модуль select
- Неблокирующий ввод/вывод
- Python фреймворки

Модуль select

- Модуль select используется для организации неблокирующего ввода/вывода.
- Существуют несколько механизмов опроса файловых дескрипторов:
  - select.select(...)
  - select.poll(...)
  - select.epoll(...)
  - select.kqueue(...)
  - ...

In [ ]:

```
# Неблокирующий ввод/вывод, обучающий пример

import socket
import select

sock = socket.socket()
sock.bind(("", 10001))
sock.listen()

# как обработать запросы для conn1 и conn2
# одновременно без потоков?
conn1, addr = sock.accept()
conn2, addr = sock.accept()

conn1.setblocking(0)
conn2.setblocking(0)

epoll = select.epoll()
epoll.register(conn1.fileno(), select.EPOLLIN | select.EPOLLO
UT)
epoll.register(conn2.fileno(), select.EPOLLIN | select.EPOLLO
UT)

conn_map = {
    conn1.fileno(): conn1,
    conn2.fileno(): conn2,
}
```

In [ ]:
```
# Неблокирующий ввод/вывод, обучающий пример
# Цикл обработки событий в epoll

while True:
    events = epoll.poll(1)

    for fileno, event in events:
        if event & select.EPOLLIN:
            # обработка чтения из сокета
            data=conn_map[fileno].recv(1024)
            print(data.decode("utf8"))
        elif event & select.EPOLLOUT:
            # обработка записи в сокет
            conn_map[fileno].send("pong".encode("utf8"))
```

In [ ]:
```
В современных ОС Linux используют epoll.
При помощи вызова epoll.poll можно получить файловые дескрипт
оры готовые для чтения или записи.
Такой код иногда называют асинхронным программированием,
или мультиплексирование ввода/вывода.
Пример сделан с целью обучения для понимания того как использ
овать неблокирующий ввод/вывод.
```

Неблокирующий ввод/вывод

- Код уже не выглядит слишком простым (хотя в нем нет создания потоков или процессов)
  - Нет обработки закрытия сокетов
  - Отсутствует обработка новых входящих соединений
- Если код будет решать настоящие задачи, то увеличится кол-во операторов if или callback-ов
- Как изменится код, если в обработке запроса появятся вызовы сторонних библиотеки?
- Не тратим память на создание процессов
- Нет накладных расходов на создание потоков и их синхронизацию

- Нет проблем с GIL
- Как спрятать вызовы select.epoll в функции библиотеки?

Фреймворки для работы в один поток

- Twisted, callback api
  - https://twistedmatrix.com (https://twistedmatrix.com)
- Gevent, greenlet, stackless python
  - http://www.gevent.org/ (http://www.gevent.org/)
- Tornado, generators api
  - http://www.tornadoweb.org (http://www.tornadoweb.org)
- Asyncio, mainstream
  - https://docs.python.org/3/library/asyncio.html (https://docs.python.org/3/library/asyncio.html)

Итераторы и генераторы, в чем разница?

- Как устроены итераторы и генераторы
- Сходства и различия

In [ ]:

```
# Итераторы

class MyRangeIterator:
    def __init__(self, top):
        self.top = top
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.current >= self.top:
            raise StopIteration

        current = self.current
        self.current += 1
        return current

>>> counter = MyRangeIterator(3)
>>> counter
<__main__.MyRangeIterator object at 0xb671b5cc>
>>> for it in counter:
>>>     print(it)
0
1
2
```

Мы рассмотрим range iterator class. Реализуем его как генератор. Метод **iter** возвращает self, созданных объект.

Затем на каждой итреции будет вызываться метод next() для этого объекта. Метод next() должен возвращать следующее значение.

Или генерировать исключение StopIteration.

In [ ]:

```
# Генераторы

def my_range_generator(top):
    current = 0
    while current < top:
        yield current
        current += 1

>>> counter = my_range_generator(3)
>>> counter
<generator object my_range_generator at 0xb67170ec>
>>> for it in counter:
>>>     print(it)
0
1
2
```

Итераторы и генераторы, подводим итоги

- Рассмотрели примеры работы итераторов и генераторов
- Они решают одну и туже задачу - генерация последовательностей
- Итератор храненит значения для следующей итерации в self
- Генератор использует локальные переменные
- В генераторы заложены большие возможности для написания concurrency кода

Генераторы и сопрограммы

- Как устроены сопрограммы?
- Отличие между генераторами и сопрограммами
- Как работает yield from?
- Примеры работы сопрограмм

In [ ]:

```
# Сопрограммы (корутины)

def grep(pattern):
    print("start grep")
    while True:
        line = yield
        if pattern in line:
            print(line)

>>> g = grep("python")
>>> next(g) # g.send(None)
start grep
>>> g.send("golang is better?")
>>> g.send("python is simple!")
python is simple!
```

In [ ]:

```
# Сопрограммы, вызов метода close()

def grep(pattern):
    print("start grep")
    try:
        while True:
            line = yield
            if pattern in line:
                print(line)
    except GeneratorExit:
        print("stop grep")

>>> g = grep("python")
>>> next(g)  # g.send(None)
start grep
>>> g.send("python is the best!")
python is the best!
>>> g.close()
stop grep
```

In [ ]:

```
# Сопрограммы, генерация исключений

def grep(pattern):
    print("start grep")
    try:
        while True:
            line = yield
            if pattern in line:
                print(line)
    except GeneratorExit:
        print("stop grep")

>>> g = grep("python")
>>> next(g) # g.send(None)
>>> g.send("python is the best!")
>>> g.throw(RuntimeError, "something wrong")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: something wrong
```

In [ ]:

```
# Вызовы сопрограмм, PEP 380

def grep(pattern):
    print("start grep")
    while True:
        line = yield
        if pattern in line:
            print(line)

def grep_python_coroutine():
    g = grep("python")
    next(g)
    g.send("python is the best!")
    g.close()

>>> g = grep_python_coroutine()  # is g coroutine?
start grep
python is the best!
>>> g
>>>
```

In [ ]:

```
# Сопрограммы, yield from PEP 0380

def grep(pattern):
    print("start grep")
    while True:
        line = yield
        if pattern in line:
            print(line)

def grep_python_coroutine():
    g = grep("python")
    yield from g

>>> g = grep_python_coroutine()  # is g coroutine?
>>> g
<generator object grep_python_coroutine at 0x7f027eec03b8>
>>> g.send(None)
start grep
>>> g.send("python wow!")
python wow!
```

In [ ]:
```
# PEP 380, генераторы

def chain(x_iterable, y_iterable):
    yield from x_iterable
    yield from y_iterable

def the_same_chain(x_iterable, y_iterable):
    for x in x_iterable:
        yield x

    for y in y_iterable:
        yield y

>>> a = [1, 2, 3]
>>> b = (4, 5)
>>> for x in chain(a, b):
...     print(x)
1
2
3
4
5
```

Генераторы и сопрограммы, подводим итоги

- Как устроены генераторы и сопрограммы
- Несмотря на некоторую схожесть, у генератора и корутины два важных отличия:
  - Генераторы "производят" значения (yield item)
  - Корутины "потребляют" значения (item = yield)
- Корутина может иметь два состояния: suspended и resumed
- yield приостанавливает корутину
- send() возобновляет работу корутины
- close() завершает выполнение

- yield from используется для делегирования вызова генератора
- Первые шаги с asyncio

Первые шаги с asyncio

- Введение в asyncio
- Примеры выполнения asyncio кода

Первые шаги с asyncio

Фреймворк asyncio это:

- часть Python3
- неблокирующий ввод/вывод
- сервисы с тысячами соединений одновременно
- в основе лежат генераторы и корутины
- линейный код, отсутствие callbacks!

In [ ]:

```
# asyncio, Hello World

import asyncio

@asyncio.coroutine
def hello_world():
    while True:
        print("Hello World!")
        yield from asyncio.sleep(1.0)

>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(hello_world())
>>> loop.close()
Hello World!
Hello World!
...
```

In [ ]:

```
# asyncio, async def / await; PEP 492 Python3.5

import asyncio

async def hello_world():
    while True:
        print("Hello World!")
        await asyncio.sleep(1.0)

>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(hello_world())
>>> loop.close()
Hello World!
Hello World!
...
```

In [ ]:

```
# asyncio, tcp сервер

import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(1024)
    message = data.decode()
    addr = writer.get_extra_info("peername")
    print("received %r from %r" % (message, addr))
    writer.close()

loop = asyncio.get_event_loop()
coro = asyncio.start_server(handle_echo, "127.0.0.1", 10001,
loop=loop)
server = loop.run_until_complete(coro)
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

In [ ]:
```
# asyncio, tcp клиент

import asyncio

async def tcp_echo_client(message, loop):
    reader, writer = await asyncio.open_connection("127.0.0.
1", 10001, loop=loop)

    print("send: %r" % message)
    writer.write(message.encode())
    writer.close()

loop = asyncio.get_event_loop()
message = "hello World!"
loop.run_until_complete(tcp_echo_client(message, loop))
loop.close()
```

Выполнение кода в asyncio

- asyncio.Future
- asyncio.Task
- loop.run_in_executor
- библиотеки для работы с asyncio

In [ ]:

```
### asyncio.Future, аналог concurrent.futures.Future

import asyncio

async def slow_operation(future):
    await asyncio.sleep(1)
    future.set_result("Future is done!")

>>> loop = asyncio.get_event_loop()
>>> future = asyncio.Future()
>>> asyncio.ensure_future(slow_operation(future))
>>>
>>> loop.run_until_complete(future)
>>> print(future.result())
Future is done!
>>> loop.close()
```

In [ ]:

```
### asyncio.Task, запуск нескольких корутин

import asyncio

async def sleep_task(num):
    for i in range(5):
        print(f"process task: {num} iter: {i}")
        await asyncio.sleep(1)

    return num

# ensure_future or create_task
>>> loop = asyncio.get_event_loop()

>>> task_list = [loop.create_task(sleep_task(i)) for i in ran
ge(2)]
>>> loop.run_until_complete(asyncio.wait(task_list))

>>> loop.run_until_complete(loop.create_task(sleep_task(3)))
>>> loop.run_until_complete(asyncio.gather(
>>>     sleep_task(10),
>>>     sleep_task(20),
>>> ))
```

In [ ]:

```
# loop.run_in_executor, запуск в отдельном потоке

import asyncio
from urllib.request import urlopen

# a synchronous function
def sync_get_url(url):
   return urlopen(url).read()

async def load_url(url, loop=None):
    future = loop.run_in_executor(None, sync_get_url, url)
    response = await future
    print(len(response))

loop = asyncio.get_event_loop()
loop.run_until_complete(load_url("https://google.com", loop=l
oop))
```

Библиотеки asyncio

- https://github.com/aio-libs (https://github.com/aio-libs)
- aiohttp
  - https://github.com/aio-libs/aiohttp (https://github.com/aio-libs/aiohttp)
- aiomysql
  - https://github.com/aio-libs/aiomysql (https://github.com/aio-libs/aiomysql)
- aiomcache
  - https://github.com/aio-libs/aiomcache (https://github.com/aio-libs/aiomcache)
- https://docs.python.org/3/library/asyncio.html (https://docs.python.org/3/library/asyncio.html)