

Tinkoff Python

Лекция 8

Асинхронное программирование



Виды соединений

Poll

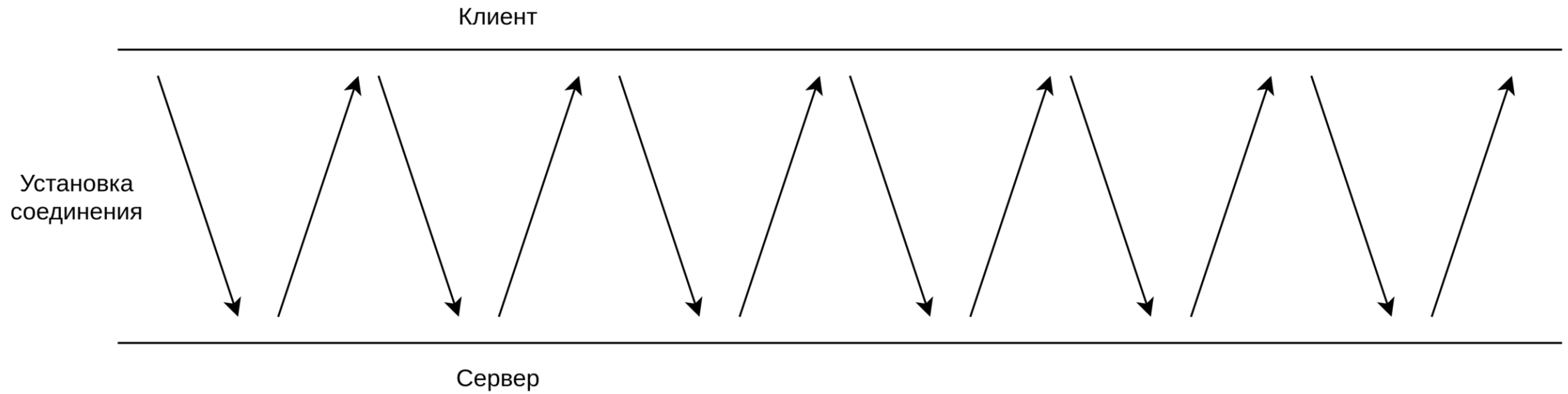
Постоянно заваливаем запросами

+ простота реализации

- очень велики расходы на постоянную
установку соединения

не рекомендуется к использованию

Poll



Long Polling

1. Отправляется запрос на сервер
2. Соединение не закрывается сервером, пока не появится событие или наступит таймаут
3. Событие отправляется в ответ на запрос
4. Клиент тут же отправляет новый ожидающий запрос

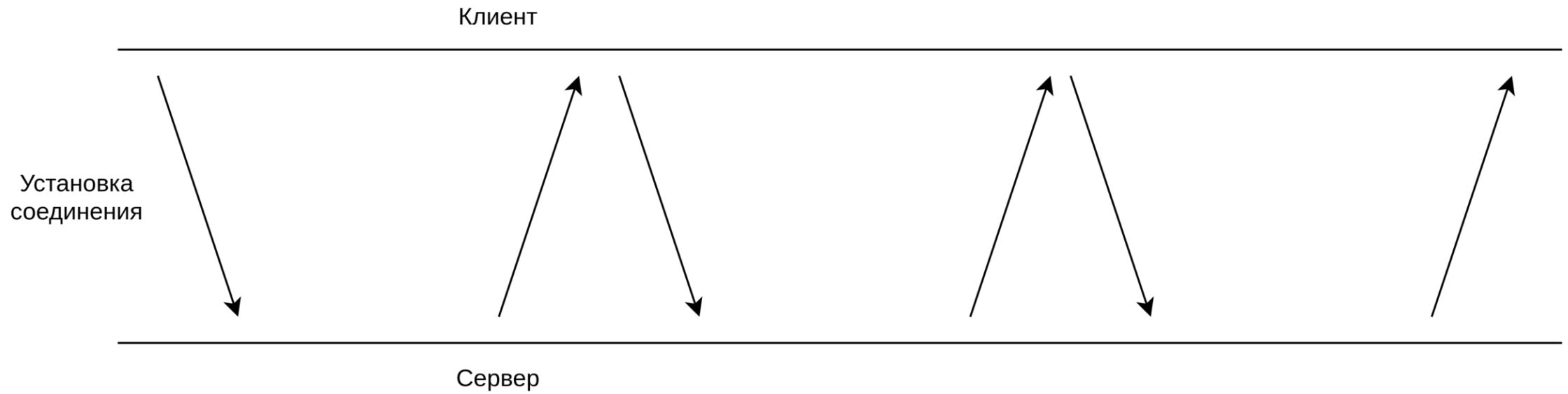
- + Простота реализации

- Опять же необходимость каждый раз устанавливать соединение

- Нет обратной связи

К примеру поможет для получения реалтайм оповещений

Long Polling

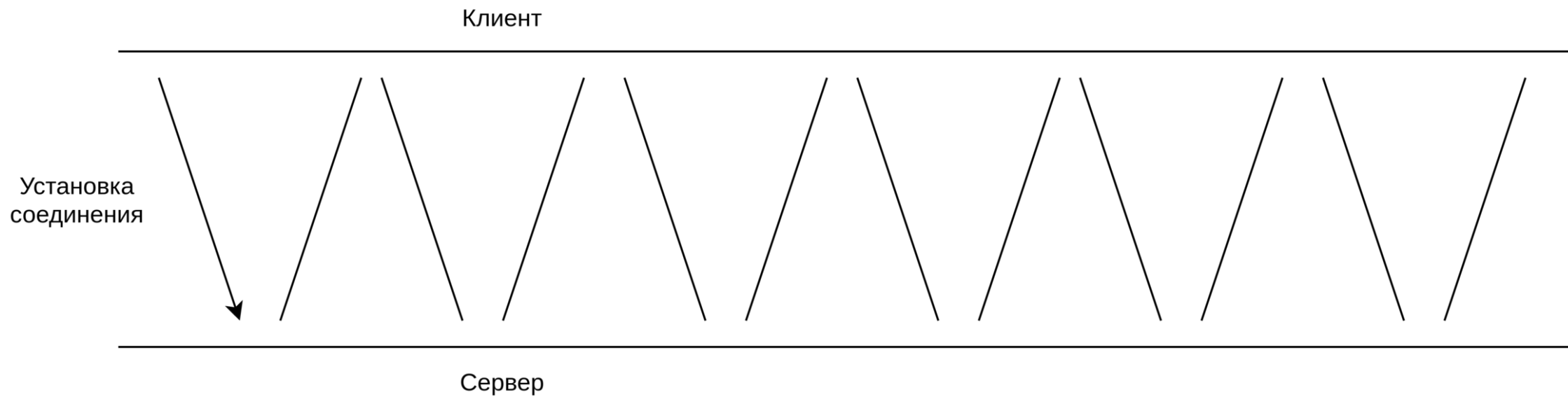


WebSocket

Соединение, при котором и клиент и сервер могут обмениваться друг с другом данными

- + Создается одно соединение и в его рамках идет передача всех данных
- + Передача данных в обе стороны
- Более сложная реализация

WebSocket



c10k

Действительно ли проблема?

c10m

вот это проблема!

10 миллионов процессов?

```
1 import time
2 from concurrent.futures import ProcessPoolExecutor
3
4 import requests
5
6 # во всех примерах используется сервер, который отвечает за 0.3 секунды
7 def load_many() -> None:
8     session = requests.Session()
9     tasks = []
10    with ProcessPoolExecutor(max_workers=100) as pool:
11        for i in range(500):
12            tasks.append(pool.submit(session.get, "http://127.0.0.1:8000/"))
13
14        for task in tasks:
15            print(task.result().json())
16
17
18 start = time.monotonic()
19 load_many()
20 duration = time.monotonic() - start
21 print(f"spend {duration} seconds")
```

```
1 python process-load.py
2 ...
3 spend 2.271605703019304 seconds
```

10 миллионов тредов?

```
1 import time
2 from concurrent.futures import ThreadPoolExecutor
3
4 import requests
5
6
7 def load_many() -> None:
8     session = requests.Session()
9     tasks = []
10    with ThreadPoolExecutor(max_workers=100) as pool:
11        for i in range(500):
12            tasks.append(pool.submit(session.get, "http://127.0.0.1:8000"))
13
14        for task in tasks:
15            print(task.result().json())
16
17
18 start = time.monotonic()
19 load_many()
20 duration = time.monotonic() - start
21 print(f"spend {duration} seconds")
```

```
1 python thread-load.py
2 ...
3 spend 2.1006762809993234 seconds
```

Слишком накладно?

Синхронизировать это все дело, следить чтобы ничего не утекло, правильно поставлены локи, потокобезопасная сессии и т.д.

На каждое пользовательское соединение отдельный поток, процесс?

Чем больше тредов, тем дольше будет переключаться контекст в операционной системе.

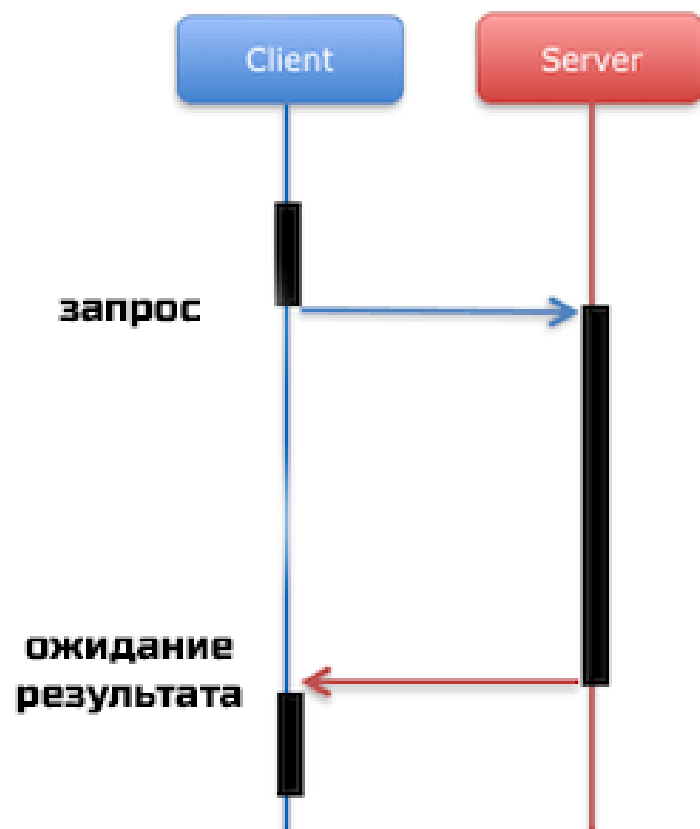
Слишком жирно?

Каждый тред или процесс будут выжирать память и время сри, при этом не использоваться на полную мощность

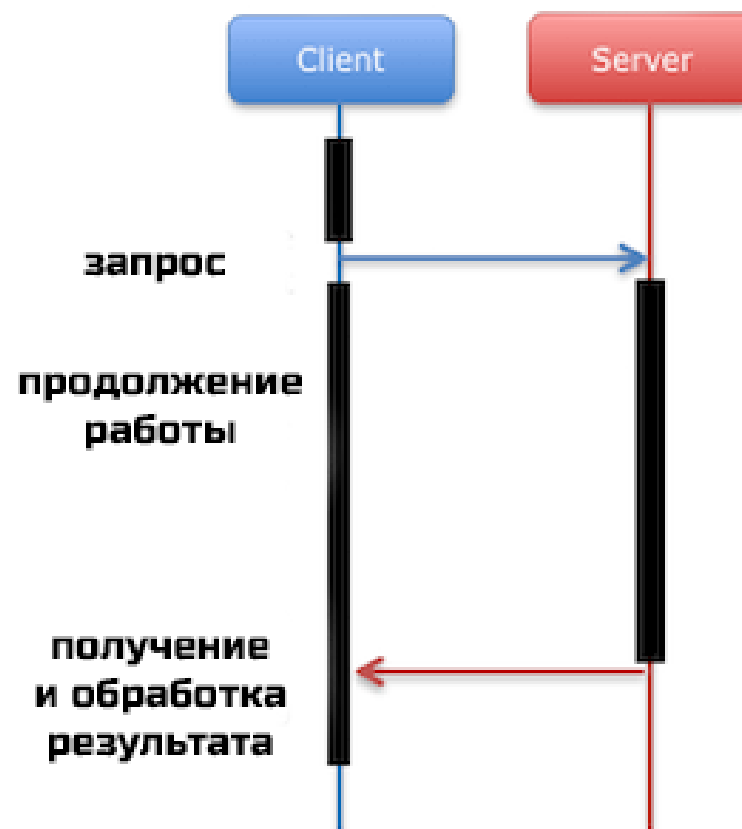
В **синхронных** операциях задачи выполняются друг за другом.

В **асинхронных** задачи могут запускаться и завершаться независимо друг от друга.

синхронно



асинхронно



Такой подход позволяет
решить проблему с
ресурсами и скоростью
работы кода

Асинхронное программирование

Это вызов асинхронных системных функций.

Но по факту сложнее.

Почему же?

Системные вызовы сложны

**Мы же пишем на
ПИТОНЕ**

Есть разные шаблоны, которые
упрощают работу с
асинхронными вызовами

Event Loop


Очередь из задач, которая регулирует порядок их запуска в системе.

Callback

Функции обратного вызова

```
1 import tornado.ioloop
2 from tornado.httpclient import AsyncHTTPClient
3
4
5 def handle_response(response):
6     if response.error:
7         print("Error:", response.error)
8     else:
9         url = response.request.url
10        data = response.body
11        print("{}: {} bytes: {}".format(url, len(data), data))
12
13
14 def load_many():
15     http_client = AsyncHTTPClient()
16     tasks = []
17     for i in range(500):
18         tasks.append(http_client.fetch("http://127.0.0.1:8000/", handle_response))
19
20     tornado.ioloop.IOLoop.instance().start()
21
22
23 load_many()
```

very painful



```
1  function hell(win) {
2    // for listener purpose
3    return function() {
4      loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
5        loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
6          loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
7            loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
8              loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {
9                loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
10                 loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
11                  loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
12                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
13                     async.eachSeries(SCRIPTS, function(src, callback) {
14                       loadScript(win, BASE_URL+src, callback);
15                     });
16                   });
17                 });
18               });
19             });
20           });
21         });
22       });
23     });
24   });
25 };
26 }
```

Можно декомпозировать и
написать иначе, но при
написании кода на колбеках
теряется линейность

Теория конечных автоматов

Конечный автомат — это некоторая абстрактная модель, содержащая конечное число состояний чего-либо, и регулирующая правило переключения этих состояний

Примитивы асинхронного кода

Корутина (сопрограмма)

Функция, которая может передавать управление циклу событий

```
1 In [1]: async def kek() -> None:
2     ...:     pass
3     ...:
4     ...: type(kek())
5     ...:
6 /home/os/.local/bin/ipython:5: RuntimeWarning: coroutine 'kek' was never awaited
7 from IPython import start_ipython
8 RuntimeWarning: Enable tracemalloc to get the object allocation traceback
9 Out[1]: coroutine
10
11 In [2]: type(kek)
12 Out[2]: function
```


await

ключевое слово для получения
результата корутины, или других
объектов типа future или task

asyncio

основная библиотека для работы с асинхронностью
является частью стандартной библиотеки

Асинхронный код не запустить вне цикла событий

```
1 import asyncio
2
3 async def main():
4     pass
5
6 loop = asyncio.get_event_loop()
7
8 loop.run_until_complete(main())
9
10 # Или просто
11 asyncio.run(main())
12
```

Task

Корутина запущенная concurrently

Что выведет этот код?

```
1 import asyncio
2
3
4 async def payload_work(num: int) -> None:
5     await asyncio.sleep(10)
6     print('kek')
7
8
9 async def main():
10     tasks = [asyncio.create_task(payload_work(i)) for i in range(10)]
11
12
13 asyncio.run(main())
```

Ничего, задача не успеет отработать

Правильно

```
1 import asyncio
2
3
4 async def payload_work(num: int) -> None:
5     await asyncio.sleep(10)
6     print('kek')
7
8
9 async def main():
10     tasks = [asyncio.create_task(payload_work(i)) for i in range(10)]
11     for task in tasks:
12         await tas
13
14 asyncio.run(main())
```

Что бы гарантировать результат
выполнения из task, необходимо ее
awaitнуть

Задачи можно отменять

Полезно при получении ошибки в коде и
задача больше не нужна

```
1 import asyncio
2
3
4 async def payload_work(num: int) -> None:
5     try:
6         await asyncio.sleep(1)
7         print(f'kek {num}')
8     except Exception as e:
9         print(e)
10
11
12 async def main():
13     tasks = [asyncio.create_task(payload_work(i)) for i in range(10)]
14     tasks[5].cancel()
15     for task in tasks:
16         await task
17
18
19 asyncio.run(main())
20 # в данном случае все задачи отработают кроме 5
```

aiohttp

один из первых асинхронных веб фреймворков на основе asyncio

```

1 # Пример программы на aiohttp, клиент
2 import asyncio
3 import time
4 from typing import Dict
5
6 import aiohttp
7
8
9 async def load_one(session: aiohttp.ClientSession) -> Dict[str, str]:
10     async with session.get("http://127.0.0.1/") as result:
11         return await result.json()
12
13
14 async def load_many() -> None:
15     tasks = []
16     async with aiohttp.ClientSession() as session:
17         for i in range(500):
18             tasks.append(asyncio.create_task(load_one(session)))
19
20     for task in tasks:
21         res = await task
22         print(res)
23
24
25 start = time.monotonic()
26 asyncio.run(load_many())
27 duration = time.monotonic() - start
28 print(duration) # 1.767098006006563

```

```
1 # пример aiohttp веб сервера
2 from aiohttp import web
3
4
5 async def hello(request):
6     return web.Response(text="Hello, world")
7
8
9 app = web.Application()
10 app.add_routes([web.get("/", hello)])
11 web.run_app(app)
```

```
1 import asyncio
2
3 from fastapi import FastAPI
4
5
6 app = FastAPI()
7
8 # тот самый сервер на котором я тестировал
9 @app.get("/")
10 async def hello_world():
11     await asyncio.sleep(0.3)
12     return {"Hello": "World"}
```

Слегка пошутим



```
1 import time
2
3 from fastapi import FastAPI
4
5
6 app = FastAPI()
7
8
9 @app.get("/")
10 async def hello_world():
11     time.sleep(0.3)
12     return {"Hello": "World"}
```

Результат стал хуже в 10 раз, хотя
сервер запущен в 10 воркеров.
асинхронная попытка выгрузить
отработала за 18 секунд, в чем же дело?

Результат стал хуже в 10 раз, хотя
сервер запущен в 10 воркеров.
асинхронная попытка выгрузить
отработала за 18 секунд, в чем же дело?

Вызывая синхронные блокирующие
операции в асинхронном приложении,
мы блокируем всё приложение.

А что если библиотека, которую мы
используем не может в асинхронный
код, но аналогов нету?


```
1 from concurrent.fututurs import ThreadPoolExecutor, ProcessPoolExecutor
2
3 await asyncio.run_in_executor(executor, function)
4
5 from functools import partial
6
7 await asyncio.run_in_executor(executor, partial(function, arg1, arg2))
```

```
1 import asyncio
2 import functools
3 import typing
4
5
6 def run_in_threadpool(func):
7     @functools.wraps(func)
8     def wrap(*args, **kwargs) -> typing.Awaitable[func]:
9         def inner():
10             return func(*args, **kwargs)
11
12         return asyncio.get_running_loop().run_in_executor(None, inner)
13
14     return wrap
15
16 @run_in_threadpool
17 def load_from_database():
18     ...
19     return res
20
21 await load_from_database()
```

**Множество библиотек имеют
асинхронные реализации**

aioredis

```
1 import asyncio
2 import aioredis
3
4 async def go():
5     redis = await aioredis.create_redis_pool(
6         'redis://localhost')
7     await redis.set('my-key', 'value')
8     val = await redis.get('my-key', encoding='utf-8')
9     print(val)
10    redis.close()
11    await redis.wait_closed()
12
13 asyncio.run(go())
14 # will print 'value'
```

pytest-asyncio

```
1 def test_http_client(event_loop):
2     url = 'http://httpbin.org/get'
3     resp = event_loop.run_until_complete(http_client(url))
4     assert b'HTTP/1.1 200 OK' in resp
5
6 @pytest.fixture()
7 async def async_fixture():
8     return await asyncio.sleep(0.1)
9
10 @pytest.mark.asyncio
11 async def test_some_asyncio_code():
12     res = await library.do_something()
13     assert b'expected result' == res
14
```

Errors

Пример 1

```
1 import asyncio
2 import time
3 from typing import Dict
4
5 import aiohttp
6
7
8 async def load_one(session: aiohttp.ClientSession) -> Dict[str, str]:
9     async with session.get("http://127.0.0.1/") as result:
10         return await result.json()
11
12
13 async def load_many() -> None:
14     tasks = []
15     async with aiohttp.ClientSession() as session:
16         for _ in range(10):
17             tasks.append(asyncio.create_task(load_one(session)))
18
19
20 start = time.monotonic()
21 asyncio.run(load_many())
22 duration = time.monotonic() - start
23 print(duration)
```

RuntimeError: Session is closed

Если задача использует контекст, необходимо проследить, что она выполнялась в рамках этого контекста

**Вывод: ждем завершения
задачи**

Пример 2

```
1 import asyncio
2
3
4 async def payload_work(num: int) -> None:
5     await asyncio.sleep(1)
6     print(f'kek {num}')
7
8
9 async def main():
10     tasks = [asyncio.create_task(payload_work(i)) for i in range(10)]
11     tasks[5].cancel()
12     try:
13         for task in tasks:
14             await task
15     except asyncio.CancelledError:
16         raise
17     except Exception:
18         print('error')
19
20
21 asyncio.run(main())
```

`asyncio.CancelledError`

Операция была отменена.

В большинстве ситуаций необходимо прокидывать дальше.

**Stop use except Exception
reraise CancelledError**

Queue

```
1 import asyncio
2
3 import aiohttp
4
5
6 COUNT = 10
7
8
9 async def fetch(session: aiohttp.ClientSession, num: int, queue: asyncio.Queue) -> None:
10     async with session.get("https://official-joke-api.appspot.com/jokes/programming/random") as result:
11         await queue.put((num, await result.json()))
12
13
14 async def handle_results(queue: asyncio.Queue) -> None:
15     while True:
16         num, data = await queue.get()
17         print((num, data))
18
19
20 async def main() -> None:
21     queue = asyncio.Queue()
22     result_task = asyncio.create_task(handle_results(queue))
23     timeout = aiohttp.ClientTimeout(total=1.2)
24     async with aiohttp.ClientSession(timeout=timeout) as session:
25         tasks = []
26         for i in range(COUNT):
27             tasks.append(asyncio.create_task(fetch(session, i, queue)))
28         for task in tasks:
29             await task
30     await result_task
31
32
33 asyncio.run(main())
```


FastAPI

<https://fastapi.tiangolo.com/>

- pydantic
- openapi
- удобство
- скорость написания кода
- хвалебные отзывы

Простой пример

```
1 from fastapi import FastAPI
2 from pydantic import BaseModel
3
4 app = FastAPI()
5
6
7 class Item(BaseModel):
8     name: str
9     price: float
10    is_offer: bool = None
11
12
13 @app.get("/")
14 async def read_root():
15     return {"Hello": "World"}
16
17
18 @app.get("/items/{item_id}")
19 async def read_item(item_id: int, q: str = None):
20     return {"item_id": item_id, "q": q}
21
22
23 @app.put("/items/{item_id}")
24 async def update_item(item_id: int, item: Item):
25     return {"item_name": item.name, "item_id": item_id}
```

- Все данные провалидируются через `pydantic`
- Автоматически сгенерируется swagger

default



GET

/ Read Root

GET

/items/{item_id} Read Item

PUT

/items/{item_id} Update Item

Schemas



HTTPValidationError >

```
Item {
  name*      string
             title: Name
  price*     number
             title: Price
  is_offer   boolean
             title: Is Offer
}
```

ValidationError >

WebSocket

```
1 from fastapi import FastAPI, WebSocket
2 from fastapi.responses import HTMLResponse
3
4 app = FastAPI()
5
6 html = """
7 <!DOCTYPE html>
8 <html>
9 ... код с js, который создает веб сокет с фронта
10 </html>
11 """
12
13
14 @app.get("/")
15 async def get():
16     return HTMLResponse(html)
17
18
19 @app.websocket("/ws")
20 async def websocket_endpoint(websocket: WebSocket):
21     await websocket.accept()
22     while True:
23         data = await websocket.receive_text()
24         await websocket.send_text(f"Message text was: {data}")
```

WebSocket Chat

- Message text was: 1
- Message text was: 2
- Message text was: 3

А если из кода?


```

1 import asyncio
2
3 import aiohttp
4
5
6 async def wc_connection():
7     async with aiohttp.ClientSession() as session:
8         async with session.ws_connect('http://127.0.0.1:8000/ws') as ws:
9             await ws.send_str('hi')
10            async for msg in ws:
11                await asyncio.sleep(1)
12                if msg.type == aiohttp.WSMsgType.TEXT:
13                    if msg.data == 'close cmd':
14                        await ws.close()
15                        break
16                    else:
17                        await ws.send_str(msg.data)
18                        print(msg.data)
19                elif msg.type == aiohttp.WSMsgType.ERROR:
20                    break
21
22
23 asyncio.run(wc_connection())

```

```

1 Message text was: hi
2 Message text was: Message text was: hi
3 Message text was: Message text was: Message text was: hi
4 Message text was: Message text was: Message text was: Message text was: hi

```

Можно писать и синхронные обработчики,

которые выполняются в `asyncio.run_in_executor`

```
1 from fastapi import FastAPI
2 from pydantic import BaseModel
3
4 app = FastAPI()
5
6
7 class Item(BaseModel):
8     name: str
9     price: float
10    is_offer: bool = None
11
12
13 @app.get("/")
14 def read_root():
15     return {"Hello": "World"}
16
17
18 @app.get("/items/{item_id}")
19 def read_item(item_id: int, q: str = None):
20     return {"item_id": item_id, "q": q}
21
22
23 @app.put("/items/{item_id}")
24 def update_item(item_id: int, item: Item):
25     return {"item_name": item.name, "item_id": item_id}
```

Тесты!

```
1 from fastapi import FastAPI
2 from fastapi.testclient import TestClient
3
4 app = FastAPI()
5
6
7 @app.get("/")
8 async def read_main():
9     return {"msg": "Hello World"}
10
11
12 client = TestClient(app)
13
14
15 def test_read_main():
16     response = client.get("/")
17     assert response.status_code == 200
18     assert response.json() == {"msg": "Hello World"}
```

WSGI -> ASGI

ASGI - духовный наследник wsgi, но
нацелен на асинхронность

uvicorn