PYTHON*

AsyncIO для практикующего python-разработчика

ПЕРЕВОД

TyVik 8 сентября 2017 в 09:27 31,4k

Оригинал: Yeray Diaz

Я помню тот момент, когда подумал «Как же медленно всё работает, что если я распараллелю вызовы?», а спустя 3 дня, взглянув на код, ничего не мог понять в жуткой каше из потоков, синхронизаторов и функций обратного вызова.

Тогда я познакомился с asyncio, и всё изменилось.

Если кто не знает, asyncio — новый модуль для организации конкурентного программирования, который появился в Python 3.4. Он предназначен для упрощения использования корутин и футур в асинхронном коде — чтобы код выглядел как синхронный, без коллбэков.

Я помню, в то время было несколько похожих инструментов, и один из них выделялся — это библиотека gevent. Я советую всем прочитать прекрасное руководство gevent для практикующего python-разработчика, в котором описана не только работа с ней, но и что такое конкурентность в общем понимании. Мне настолько понравилось та статья, что я решил использовать её как шаблон для написания введения в asyncio.

Небольшой дисклеймер — это статья не gevent vs asyncio. Nathan Road уже сделал это за меня в своей заметке. Все примеры вы можете найти на GitHub.

Я знаю, вам уже не терпится писать код, но для начала я бы хотел рассмотреть несколько концепций, которые нам пригодятся в дальнейшем.

Потоки, циклы событий, корутины и футуры

Потоки — наиболее распространённый инструмент. Думаю, вы слышали о нём и ранее, однако asyncio оперирует несколько другими понятиями: циклы событий, корутины и футуры.

- цикл событий (event loop) по большей части всего лишь управляет выполнением различных задач: регистрирует поступление и запускает в подходящий момент
- корутины — специальные функции, похожие на генераторы python, от которых ожидают (**await**), что они будут отдавать управление обратно в цикл событий. Необходимо, чтобы они были запущены именно через цикл событий
- футуры — объекты, в которых хранится текущий результат выполнения какой-либо задачи. Это может быть информация о том, что задача ещё не обработана или уже полученный результат; а может быть вообще исключение

Довольно просто? Поехали!

Синхронное и асинхронное выполнение

В видео "Конкурентность — это не параллелизм, это лучше" Роб Пайк обращает ваше внимание на ключевую вещь. Разбиение задач на конкурентные подзадачи возможно только при таком параллелизме, когда он же и управляет этими подзадачами.

Asyncio делает тоже самое — вы можете разбивать ваш код на процедуры, которые определять как корутины, что даёт возможность управлять ими как пожелаете, включая и одновременное выполнение. Корутины содержат операторы yield, с помощью которых мы определяем места, где можно переключиться на другие ожидающие выполнения задачи.

За переключение контекста в asyncio отвечает yield, который передаёт управление обратно в event loop, а тот в свою очередь — к другой корутине. Рассмотрим базовый пример:

```
import asyncio

async def foo():
    print('Running in foo')
    await asyncio.sleep(0)
    print('Explicit context switch to foo again')


async def bar():
    print('Explicit context to bar')
    await asyncio.sleep(0)
    print('Implicit context switch back to bar')


ioloop = asyncio.get_event_loop()
tasks = [ioloop.create_task(foo()), ioloop.create_task(bar())]
wait_tasks = asyncio.wait(tasks)
ioloop.run_until_complete(wait_tasks)
ioloop.close()
```

```
$ python3 1-sync-async-execution-asyncio-await.py
Running in foo
Explicit context to bar
Explicit context switch to foo again
Implicit context switch back to bar
```

* Сначала мы объявили пару простейших корутин, которые притворяются неблокирующими, используя **sleep** из asyncio
* Корутины могут быть запущены только из другой корутины, или обёрнуты в задачу с помощью **create_task**
* После того, как у нас оказались 2 задачи, объединим их, используя **wait**
* И, наконец, отправим на выполнение в цикл событий через **run_until_complete**

Используя **await** в какой-либо корутине, мы таким образом объявляем, что корутина может отдавать управление обратно в event loop, который, в свою очередь, запустит какую-либо следующую задачу: bar. В bar произойдёт тоже самое: на **await asyncio.sleep** управление будет передано обратно в цикл событий, который в нужное время вернётся к выполнению foo.

Представим 2 блокирующие задачи: gr1 и gr2, как будто они обращаются к неким сторонним сервисам, и, пока они ждут ответа, третья функция может работать асинхронно.

```
import time
import asyncio

start = time.time()
```

```
def tic():
    return 'at %1.1f seconds' % (time.time() - start)


async def gr1():
    # Busy waits for a second, but we don't want to stick around...
    print('gr1 started work: {}'.format(tic()))
    await asyncio.sleep(2)
    print('gr1 ended work: {}'.format(tic()))


async def gr2():
    # Busy waits for a second, but we don't want to stick around...
    print('gr2 started work: {}'.format(tic()))
    await asyncio.sleep(2)
    print('gr2 Ended work: {}'.format(tic()))


async def gr3():
    print("Let's do some stuff while the coroutines are blocked,
{}".format(tic()))
    await asyncio.sleep(1)
    print("Done!")


ioloop = asyncio.get_event_loop()
tasks = [
    ioloop.create_task(gr1()),
    ioloop.create_task(gr2()),
    ioloop.create_task(gr3())
]
ioloop.run_until_complete(asyncio.wait(tasks))
ioloop.close()
```

```
$ python3 1b-cooperatively-scheduled-asyncio-await.py
gr1 started work: at 0.0 seconds
gr2 started work: at 0.0 seconds
Lets do some stuff while the coroutines are blocked, at 0.0 seconds
Done!
gr1 ended work: at 2.0 seconds
gr2 Ended work: at 2.0 seconds
```

Обратите внимание как происходит работа с вводом-выводом и

планированием выполнения, позволяя всё это уместить в один поток. Пока две задачи заблокированы ожиданием I/O, третья функция может занимать всё процессорное время.

Порядок выполнения

В синхронном мире мы мыслим последовательно. Если у нас есть список задач, выполнение которых занимает разное время, то они завершатся в том же порядке, в котором поступили в обработку. Однако, в случае конкурентности нельзя быть в этом уверенным.

```
import random
from time import sleep
import asyncio


def task(pid):
    """Synchronous non-deterministic task.
    """
    sleep(random.randint(0, 2) * 0.001)
    print('Task %s done' % pid)


async def task_coro(pid):
    """Coroutine non-deterministic task
    """
    await asyncio.sleep(random.randint(0, 2) * 0.001)
    print('Task %s done' % pid)


def synchronous():
    for i in range(1, 10):
        task(i)


async def asynchronous():
    tasks = [asyncio.ensure_future(task_coro(i)) for i in range(1,
10)]
    await asyncio.wait(tasks)
```

```
print('Synchronous:')
synchronous()

ioloop = asyncio.get_event_loop()
print('Asynchronous:')
ioloop.run_until_complete(asynchronous())
ioloop.close()
```

```
$ python3 1c-determinism-sync-async-asyncio-await.py
Synchronous:
Task 1 done
Task 2 done
Task 3 done
Task 4 done
Task 5 done
Task 6 done
Task 7 done
Task 8 done
Task 9 done
Asynchronous:
Task 2 done
Task 5 done
Task 6 done
Task 8 done
Task 9 done
Task 1 done
Task 4 done
Task 3 done
Task 7 done
```

Разумеется, ваш результат будет иным, поскольку каждая задача будет засыпать на случайное время, но заметьте, что результат выполнения полностью отличается, хотя мы всегда ставим задачи в одном и том же порядке.

Также обратите внимание на корутину для нашей довольно простой задачи. Это важно для понимания, что в asyncio нет никакой магии при реализации неблокирующих задач. Во время реализации asyncio стоял отдельно в стандартной библиотеке, т.к.

остальные модули предоставляли только блокирующую функциональность. Вы можете использовать модуль concurrent.futures для оборачивания блокирующих задач в потоки или процессы и получения футуры для использования в asyncio. Несколько таких примеров доступны на GitHub.
Это, наверно, главный недостаток сейчас при использовании asyncio, однако уже есть несколько библиотек, помогающих решить эту проблему.

Самая популярная блокирующая задача — получение данных по HTTP-запросу. Рассмотрим работу с великолепной библиотекой aiohttp на примере получения информации о публичных событиях на GitHub.

```
import time
import urllib.request
import asyncio
import aiohttp

URL = 'https://api.github.com/events'
MAX_CLIENTS = 3


def fetch_sync(pid):
    print('Fetch sync process {} started'.format(pid))
    start = time.time()
    response = urllib.request.urlopen(URL)
    datetime = response.getheader('Date')

    print('Process {}: {}, took: {:.2f} seconds'.format(
        pid, datetime, time.time() - start))

    return datetime


async def fetch_async(pid):
    print('Fetch async process {} started'.format(pid))
    start = time.time()
    response = await aiohttp.request('GET', URL)
```

```
    datetime = response.headers.get('Date')

    print('Process {}: {}, took: {:.2f} seconds'.format(
        pid, datetime, time.time() - start))

    response.close()
    return datetime


def synchronous():
    start = time.time()
    for i in range(1, MAX_CLIENTS + 1):
        fetch_sync(i)
    print("Process took: {:.2f} seconds".format(time.time() -
start))


async def asynchronous():
    start = time.time()
    tasks = [asyncio.ensure_future(
        fetch_async(i)) for i in range(1, MAX_CLIENTS + 1)]
    await asyncio.wait(tasks)
    print("Process took: {:.2f} seconds".format(time.time() -
start))


print('Synchronous:')
synchronous()

print('Asynchronous:')
ioloop = asyncio.get_event_loop()
ioloop.run_until_complete(asynchronous())
ioloop.close()
```

```
$ python3 1d-async-fetch-from-server-asyncio-await.py
Synchronous:
Fetch sync process 1 started
Process 1: Wed, 17 Feb 2016 13:10:11 GMT, took: 0.54 seconds
Fetch sync process 2 started
Process 2: Wed, 17 Feb 2016 13:10:11 GMT, took: 0.50 seconds
Fetch sync process 3 started
Process 3: Wed, 17 Feb 2016 13:10:12 GMT, took: 0.48 seconds
Process took: 1.54 seconds
Asynchronous:
Fetch async process 1 started
Fetch async process 2 started
```

```
Fetch async process 3 started
Process 3: Wed, 17 Feb 2016 13:10:12 GMT, took: 0.50 seconds
Process 2: Wed, 17 Feb 2016 13:10:12 GMT, took: 0.52 seconds
Process 1: Wed, 17 Feb 2016 13:10:12 GMT, took: 0.54 seconds
Process took: 0.54 seconds
```

Тут стоит обратить внимание на пару моментов.

Во-первых, разница во времени — при использовании асинхронных вызовов мы запускаем запросы одновременно. Как говорилось ранее, каждый из них передавал управление следующему и возвращал результат по завершении. То есть скорость выполнения напрямую зависит от времени работы самого медленного запроса, который занял как раз 0.54 секунды. Круто, правда?

Во-вторых, насколько код похож на синхронный. Это же по сути одно и то же! Основные отличия связаны с реализацией библиотеки для выполнения запросов, созданием и ожиданием завершения задач.

Создание конкурентности

До сих пор мы использовали единственный метод создания и получения результатов из корутин, создания набора задач и ожидания их завершения. Однако, корутины могут быть запланированы для запуска и получения результатов несколькими способами. Представьте ситуацию, когда нам надо обрабатывать результаты GET-запросов по мере их получения; на самом деле реализация очень похожа на предыдущую:

```
import time
import random
import asyncio
import aiohttp

URL = 'https://api.github.com/events'
MAX_CLIENTS = 3


async def fetch_async(pid):
    start = time.time()
    sleepy_time = random.randint(2, 5)
    print('Fetch async process {} started, sleeping for {}
seconds'.format(
        pid, sleepy_time))

    await asyncio.sleep(sleepy_time)

    response = await aiohttp.request('GET', URL)
    datetime = response.headers.get('Date')

    response.close()
    return 'Process {}: {}, took: {:.2f} seconds'.format(
        pid, datetime, time.time() - start)


async def asynchronous():
    start = time.time()
    futures = [fetch_async(i) for i in range(1, MAX_CLIENTS + 1)]
    for i, future in enumerate(asyncio.as_completed(futures)):
        result = await future
        print('{} {}'.format(">>" * (i + 1), result))

    print("Process took: {:.2f} seconds".format(time.time() -
start))


ioloop = asyncio.get_event_loop()
ioloop.run_until_complete(asynchronous())
ioloop.close()
```

```
$ python3 2a-async-fetch-from-server-as-completed-asyncio-await.py
Fetch async process 1 started, sleeping for 4 seconds
Fetch async process 3 started, sleeping for 5 seconds
Fetch async process 2 started, sleeping for 3 seconds
```

```
>> Process 2: Wed, 17 Feb 2016 13:55:19 GMT, took: 3.53 seconds
>>>> Process 1: Wed, 17 Feb 2016 13:55:20 GMT, took: 4.49 seconds
>>>>>> Process 3: Wed, 17 Feb 2016 13:55:21 GMT, took: 5.48 seconds
Process took: 5.48 seconds
```

Посмотрите на отступы и тайминги — мы запустили все задачи одновременно, однако они обработаны в порядке завершения выполнения. Код в данном случае немного отличается: мы пакуем корутины, каждая из которых уже подготовлена для выполнения, в список. Функция **as_completed** возвращает итератор, который выдаёт результаты корутин по мере их выполнения. Круто же, правда?! Кстати, и **as_completed**, и **wait** — функции из пакета **concurrent.futures**.

Ещё один пример — что если вы хотите узнать свой IP адрес. Есть куча сервисов для этого, но вы не знаете какой из них будет доступен в момент работы программы. Вместо того, чтобы последовательно опрашивать каждый из списка, можно запустить все запросы конкурентно и выбрать первый успешный.

Что ж, для этого в нашей любимой функции **wait** есть специальный параметр **return_when**. До сих пор мы игнорировали то, что возвращает **wait**, т.к. только распараллеливали задачи. Но теперь нам надо получить результат из корутины, так что будем использовать набор футур done и pending.

```
from collections import namedtuple
import time
import asyncio
from concurrent.futures import FIRST_COMPLETED
import aiohttp

Service = namedtuple('Service', ('name', 'url', 'ip_attr'))
```

```
SERVICES = (
    Service('ipify', 'https://api.ipify.org?format=json', 'ip'),
    Service('ip-api', 'http://ip-api.com/json', 'query')
)


async def fetch_ip(service):
    start = time.time()
    print('Fetching IP from {}'.format(service.name))

    response = await aiohttp.request('GET', service.url)
    json_response = await response.json()
    ip = json_response[service.ip_attr]

    response.close()
    return '{} finished with result: {}, took: {:.2f}
seconds'.format(
        service.name, ip, time.time() - start)


async def asynchronous():
    futures = [fetch_ip(service) for service in SERVICES]
    done, pending = await asyncio.wait(
        futures, return_when=FIRST_COMPLETED)

    print(done.pop().result())


ioloop = asyncio.get_event_loop()
ioloop.run_until_complete(asynchronous())
ioloop.close()
```

```
$ python3 2c-fetch-first-ip-address-response-await.py
Fetching IP from ip-api
Fetching IP from ipify
ip-api finished with result: 82.34.76.170, took: 0.09 seconds
Unclosed client session
client_session: <aiohttp.client.ClientSession object at 0x10f95c6d8>
Task was destroyed but it is pending!
task: <Task pending coro=<fetch_ip() running at 2c-fetch-first-ip-
address-response.py:20> wait_for=<Future pending cb=
[BaseSelectorEventLoop._sock_connect_done(10)(), Task._wakeup()]>>
```

Что же случилось? Первый сервис ответил успешно, но в логах какое-то предупреждение!

На самом деле мы запустили выполнение двух задач, но вышли из цикла уже после первого результата, в то время как вторая корутина ещё выполнялась. Asyncio подумал что это баг и предупредил нас. Наверно, стоит прибираться за собой и явно убивать ненужные задачи. Как? Рад, что вы спросили.

Состояния футур

- ожидание (pending)
- выполнение (running)
- выполнено (done)
- отменено (cancelled)

Всё настолько просто. Когда футура находится в состояние done, у неё можно получить результат выполнения. В состояниях pending и running такая операция приведёт к исключению **InvalidStateError**, а в случае canelled будет **CancelledError**, и наконец, если исключение произошло в самой корутине, оно будет сгенерировано снова (также, как это сделано при вызове **exception**). Но не верьте мне на слово.

Вы можете узнать состояние футуры с помощью методов **done**, **cancelled** или **running**, но не забывайте, что в случае **done** вызов **result** может вернуть как ожидаемый результат, так и исключение, которое возникло в процессе работы. Для отмены выполнения

футуры есть метод **cancel**. Это подходит для исправления нашего примера.

```
from collections import namedtuple
import time
import asyncio
from concurrent.futures import FIRST_COMPLETED
import aiohttp

Service = namedtuple('Service', ('name', 'url', 'ip_attr'))

SERVICES = (
    Service('ipify', 'https://api.ipify.org?format=json', 'ip'),
    Service('ip-api', 'http://ip-api.com/json', 'query')
)


async def fetch_ip(service):
    start = time.time()
    print('Fetching IP from {}'.format(service.name))

    response = await aiohttp.request('GET', service.url)
    json_response = await response.json()
    ip = json_response[service.ip_attr]

    response.close()
    return '{} finished with result: {}, took: {:.2f}
seconds'.format(
        service.name, ip, time.time() - start)


async def asynchronous():
    futures = [fetch_ip(service) for service in SERVICES]
    done, pending = await asyncio.wait(
        futures, return_when=FIRST_COMPLETED)

    print(done.pop().result())

    for future in pending:
        future.cancel()


ioloop = asyncio.get_event_loop()
ioloop.run_until_complete(asynchronous())
ioloop.close()
```

```
$ python3 2c-fetch-first-ip-address-response-no-warning-await.py
Fetching IP from ipify
Fetching IP from ip-api
ip-api finished with result: 82.34.76.170, took: 0.08 seconds
```

Простой и аккуратный вывод — как раз то, что я люблю!

Если вам нужна некоторая дополнительная логика по обработке футур, то вы можете подключать коллбэки, которые будут вызваны при переходе в состояние done. Это может быть полезно для тестов, когда некоторые результаты надо переопределить какими-то своими значениями.

Обработка исключений

asyncio — это целиком про написание управляемого и читаемого конкурентного кода, что хорошо заметно при обработке исключений. Вернёмся к примеру, чтобы продемонстрировать. Допустим, мы хотим убедиться, что все запросы к сервисам по определению IP вернули одинаковый результат. Однако, один из них может быть оффлайн и не ответить нам. Просто применим try...except как обычно:

```
from collections import namedtuple
import time
import asyncio
import aiohttp

Service = namedtuple('Service', ('name', 'url', 'ip_attr'))

SERVICES = (
    Service('ipify', 'https://api.ipify.org?format=json', 'ip'),
```

```
    Service('ip-api', 'http://ip-api.com/json', 'query'),
    Service('borken', 'http://no-way-this-is-going-to-
work.com/json', 'ip')
)


async def fetch_ip(service):
    start = time.time()
    print('Fetching IP from {}'.format(service.name))

    try:
        response = await aiohttp.request('GET', service.url)
    except:
        return '{} is unresponsive'.format(service.name)

    json_response = await response.json()
    ip = json_response[service.ip_attr]

    response.close()
    return '{} finished with result: {}, took: {:.2f}
seconds'.format(
        service.name, ip, time.time() - start)


async def asynchronous():
    futures = [fetch_ip(service) for service in SERVICES]
    done, _ = await asyncio.wait(futures)

    for future in done:
        print(future.result())


ioloop = asyncio.get_event_loop()
ioloop.run_until_complete(asynchronous())
ioloop.close()
```

```
$ python3 3a-fetch-ip-addresses-fail-await.py
Fetching IP from ip-api
Fetching IP from borken
Fetching IP from ipify
ip-api finished with result: 85.133.69.250, took: 0.75 seconds
ipify finished with result: 85.133.69.250, took: 1.37 seconds
borken is unresponsive
```

Мы также можем обработать исключение, которое возникло в процессе выполнения корутины:

```
from collections import namedtuple
import time
import asyncio
import aiohttp
import traceback

Service = namedtuple('Service', ('name', 'url', 'ip_attr'))

SERVICES = (
    Service('ipify', 'https://api.ipify.org?format=json', 'ip'),
    Service('ip-api', 'http://ip-api.com/json', 'this-is-not-an-
attr'),
    Service('borken', 'http://no-way-this-is-going-to-
work.com/json', 'ip')
)


async def fetch_ip(service):
    start = time.time()
    print('Fetching IP from {}'.format(service.name))

    try:
        response = await aiohttp.request('GET', service.url)
    except:
        return '{} is unresponsive'.format(service.name)

    json_response = await response.json()
    ip = json_response[service.ip_attr]

    response.close()
    return '{} finished with result: {}, took: {:.2f}
seconds'.format(
        service.name, ip, time.time() - start)


async def asynchronous():
    futures = [fetch_ip(service) for service in SERVICES]
    done, _ = await asyncio.wait(futures)

    for future in done:
        try:
```

```
            print(future.result())
        except:
            print("Unexpected error:
{}".format(traceback.format_exc()))


ioloop = asyncio.get_event_loop()
ioloop.run_until_complete(asynchronous())
ioloop.close()
```

```
$ python3 3b-fetch-ip-addresses-future-exceptions-await.py
Fetching IP from ipify
Fetching IP from borken
Fetching IP from ip-api
ipify finished with result: 85.133.69.250, took: 0.91 seconds
borken is unresponsive
Unexpected error: Traceback (most recent call last):
 File “3b-fetch-ip-addresses-future-exceptions.py”, line 39, in
asynchronous
 print(future.result())
 File “3b-fetch-ip-addresses-future-exceptions.py”, line 26, in
fetch_ip
 ip = json_response[service.ip_attr]
KeyError: ‘this-is-not-an-attr’
```

Точно также, как и запуск задачи без ожидания её завершения является ошибкой, так и получение неизвестных исключений оставляет свои следы в выводе:

```
from collections import namedtuple
import time
import asyncio
import aiohttp

Service = namedtuple('Service', ('name', 'url', 'ip_attr'))

SERVICES = (
    Service('ipify', 'https://api.ipify.org?format=json', 'ip'),
    Service('ip-api', 'http://ip-api.com/json', 'this-is-not-an-
attr'),
    Service('borken', 'http://no-way-this-is-going-to-
work.com/json', 'ip')
```

```
)


async def fetch_ip(service):
    start = time.time()
    print('Fetching IP from {}'.format(service.name))

    try:
        response = await aiohttp.request('GET', service.url)
    except:
        print('{} is unresponsive'.format(service.name))
    else:
        json_response = await response.json()
        ip = json_response[service.ip_attr]

        response.close()
        print('{} finished with result: {}, took: {:.2f}
seconds'.format(
            service.name, ip, time.time() - start))


async def asynchronous():
    futures = [fetch_ip(service) for service in SERVICES]
    await asyncio.wait(futures)  # intentionally ignore results


ioloop = asyncio.get_event_loop()
ioloop.run_until_complete(asynchronous())
ioloop.close()
```

```
$ python3 3c-fetch-ip-addresses-ignore-exceptions-await.py
Fetching IP from ipify
Fetching IP from borken
Fetching IP from ip-api
borken is unresponsive
ipify finished with result: 85.133.69.250, took: 0.78 seconds
Task exception was never retrieved
future: <Task finished coro=<fetch_ip() done, defined at 3c-fetch-
ip-addresses-ignore-exceptions.py:15> exception=KeyError('this-is-
not-an-attr',)>
Traceback (most recent call last):
 File "3c-fetch-ip-addresses-ignore-exceptions.py", line 25, in
fetch_ip
 ip = json_response[service.ip_attr]
KeyError: 'this-is-not-an-attr'
```

Вывод выглядит также, как и в предыдущем примере за исключением укоризненного сообщения от asyncio.

Таймауты

А что, если информация о нашем IP не так уж важна? Это может быть хорошим дополнением к какому-то составному ответу, в котором эта часть будет опциональна. В таком случае не будем заставлять пользователя ждать. В идеале мы бы ставили таймаут на вычисление IP, после которого в любом случае отдавали ответ пользователю, даже без этой информации.

И снова у **wait** есть подходящий аргумент:

```
import time
import random
import asyncio
import aiohttp
import argparse
from collections import namedtuple
from concurrent.futures import FIRST_COMPLETED

Service = namedtuple('Service', ('name', 'url', 'ip_attr'))

SERVICES = (
    Service('ipify', 'https://api.ipify.org?format=json', 'ip'),
    Service('ip-api', 'http://ip-api.com/json', 'query'),
)

DEFAULT_TIMEOUT = 0.01


async def fetch_ip(service):
    start = time.time()
    print('Fetching IP from {}'.format(service.name))

    await asyncio.sleep(random.randint(1, 3) * 0.1)
```

```
    try:
        response = await aiohttp.request('GET', service.url)
    except:
        return '{} is unresponsive'.format(service.name)

    json_response = await response.json()
    ip = json_response[service.ip_attr]

    response.close()
    print('{} finished with result: {}, took: {:.2f}
seconds'.format(
        service.name, ip, time.time() - start))
    return ip


async def asynchronous(timeout):
    response = {
        "message": "Result from asynchronous.",
        "ip": "not available"
    }

    futures = [fetch_ip(service) for service in SERVICES]
    done, pending = await asyncio.wait(
        futures, timeout=timeout, return_when=FIRST_COMPLETED)

    for future in pending:
        future.cancel()

    for future in done:
        response["ip"] = future.result()

    print(response)


parser = argparse.ArgumentParser()
parser.add_argument(
    '-t', '--timeout',
    help='Timeout to use, defaults to {}'.format(DEFAULT_TIMEOUT),
    default=DEFAULT_TIMEOUT, type=float)
args = parser.parse_args()

print("Using a {} timeout".format(args.timeout))
ioloop = asyncio.get_event_loop()
ioloop.run_until_complete(asynchronous(args.timeout))
ioloop.close()
```

Я также добавил аргумент timeout к строке запуска скрипта, чтобы проверить что же произойдёт, если запросы успеют обработаться. Также я добавил случайные задержки, чтобы скрипт не завершался слишком быстро, и было время разобраться как именно он работает.

```
$ python 4a-timeout-with-wait-kwarg-await.py
Using a 0.01 timeout
Fetching IP from ipify
Fetching IP from ip-api
{‘message’: ‘Result from asynchronous.’, ‘ip’: ‘not available’}
```

```
$ python 4a-timeout-with-wait-kwarg-await.py -t 5
Using a 5.0 timeout
Fetching IP from ip-api
Fetching IP from ipify
ipify finished with result: 82.34.76.170, took: 1.24 seconds
{'ip': '82.34.76.170', 'message': 'Result from asynchronous.'}
```

Заключение

Asyncio укрепил мою и так уже большую любовь к python. Если честно, я влюбился в сопрограммы, ещё когда познакомился с ними в Tornado, но asyncio сумел взять всё лучшее из него и других библиотек по реализации конкурентности. Причём настолько, что были предприняты особые усилия, чтобы они могли использовать основной цикл ввода-вывода. Так что если вы используете Tornado или Twisted, то можете подключать код, предназначенный для asyncio!

Как я уже упоминал, основная проблема заключается в том, что стандартные библиотеки пока ещё не поддерживают неблокирующее поведение. Также и многие популярные библиотеки работают пока лишь в синхронном стиле, а те, что используют конкурентность, пока ещё молоды и экспериментальны. Однако, их число растёт.

Надеюсь, в этом уроке я показал, насколько приятно работать с asyncio, и эта технология подтолкнёт вас к переходу на python 3, если вы по какой-то причине застряли на python 2.7. Одно точно — будущее Python полностью изменилось.

От переводчика:

Окружение для разработки на aiorest (asyncio) + angular.js

6

lmbolc • 13 августа 2014 в 12:46

Примеры использования asyncio: HTTPServer?!

9

Alesh • 26 марта 2014 в 19:27

Python на примере демона уведомления о новых коммитах Git

27

ИЗ ПЕСОЧНИЦЫ

afschr • 28 декабря 2011 в 02:10

Популярное за сутки

Яндекс открывает Алису для всех разработчиков. Платформа Яндекс.Диалоги (бета)

69

BarakAdama • вчера в 10:52

Почему следует игнорировать истории основателей успешных стартапов

20

ПЕРЕВОД

m1rko • вчера в 10:44

Как получить телефон (почти) любой красотки в Москве, или интересная особенность MT_FREE

24

ИЗ ПЕСОЧНИЦЫ

cab404 • вчера в 20:27

Java и Project Reactor

zealot_and_frenzy • вчера в 10:56

10

Пользовательские агрегатные и оконные функции в PostgreSQL и Oracle

erogov • вчера в 12:46

6

Лучшее на Geektimes

Как фермеры Дикого Запада организовали телефонную сеть на колючей проволоке

NAGru • вчера в 10:10

31

Энтузиаст сделал новую материнскую плату для ThinkPad X200s

alizar • вчера в 15:32

49

Кто-то посылает секс-игрушки с Amazon незнакомцам. Amazon не знает, как их остановить

Pochtoycom • вчера в 13:06

85

Илон Маск продолжает убеждать в необходимости создания колонии людей на Марсе

marks • вчера в 14:19

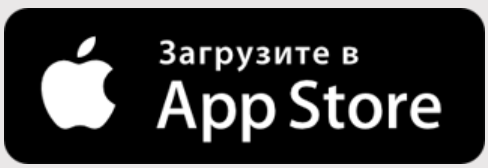
140

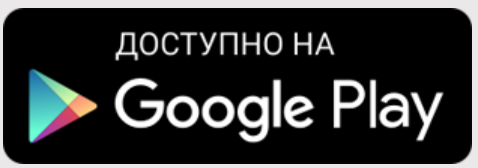
Дела шпионские (часть 1)

16

TashaFridrih • вчера в 13:16

Мобильное приложение





Полная версия