Асинхронное программирование в Python

Источник: https://tproger.ru/translations/asynchronous-programming-in-python/



Асинхронное программирование на Python становится все более популярным. Для этих целей существует множество различных библиотек. Самая популярная из них — [Asyncio](https://docs.python.org/3/library/asyncio.html" \t "_blank), которая является стандартной библиотекой Python 3.4. Из этой статьи вы узнаете, что такое асинхронное программирование и чем отличаются различные библиотеки, реализующие асинхронность в Python.

По очереди

В каждой программе строки кода выполняются поочередно. Например, если у вас есть строка кода, которая запрашивает что-либо с сервера, то это означает, что ваша программа не делает ничего во время ожидания ответа. В некоторых случаях это допустимо, но во многих — нет. Одним из решений этой проблемы являются потоки (threads).

Потоки дают возможность вашей программе выполнять ряд задач одновременно. Конечно, у потоков есть ряд недостатков. Многопоточные программы являются более сложными и, как правило, более подвержены ошибкам. Они включают в себя такие проблемы: [состояние гонки](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D0%B5_%D0%B3%D0%BE%D0%BD%D0%BA%D0%B8) (race condition), взаимная (deadlock) и активная (livelock) [блокировка,](https://ru.wikipedia.org/wiki/%D0%92%D0%B7%D0%B0%D0%B8%D0%BC%D0%BD%D0%B0%D1%8F_%D0%B1%D0%BB%D0%BE%D0%BA%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0) [исчерпание ресурсов](https://en.wikipedia.org/wiki/Starvation_(computer_science)) (resource starvation).

Переключение контекста

Хотя асинхронное программирование и позволяет обойти проблемные места потоков, оно было разработано для совершенно другой цели — для переключения контекста процессора. Когда у вас есть несколько потоков, каждое ядро процессора может запускать только один поток за раз. Для того, чтобы все потоки/процессы могли совместно использовать ресурсы, процессор очень часто переключает контекст. Чтобы упростить работу, процессор с произвольной периодичностью сохраняет всю контекстную информацию потока и переключается на другой поток.

Асинхронное программирование — это потоковая обработка программного обеспечения / пользовательского пространства, где приложение, а не процессор, управляет потоками и переключением контекста. В асинхронном программировании контекст переключается только в заданных точках переключения, а не с периодичностью, определенной CPU.

Эффективный секретарь

Теперь давайте рассмотрим эти понятия на примерах из жизни. Представьте секретаря, который настолько эффективен, что не тратит время впустую. У него есть пять заданий, которые он выполняет одновременно: отвечает на телефонные звонки, принимает посетителей, пытается забронировать билеты на самолет, контролирует графики встреч и заполняет документы. Теперь представьте, что такие задачи, как контроль графиков встреч, прием телефонных звонков и посетителей, повторяются не часто и распределены во времени. Таким образом, большую часть времени секретарь разговаривает по телефону с авиакомпанией, заполняя при этом документы. Это легко представить. Когда поступит телефонный звонок, он поставит разговор с авиакомпанией на паузу, ответит на звонок, а затем вернется к разговору с авиакомпанией. В любое время, когда новая задача потребует внимания секретаря, заполнение документов будет отложено, поскольку оно не критично. Секретарь, выполняющий несколько задач одновременно, переключает контекст в нужное ему время. Он асинхронный.

Потоки — это пять секретарей, у каждого из которых по одной задаче, но только одному из них разрешено работать в определенный момент времени. Для того, чтобы секретари работали в потоковом режиме, необходимо устройство, которое контролирует их работу, но ничего не понимает в самих задачах. Поскольку устройство не понимает характер задач, оно постоянно переключалось бы между пятью секретарями, даже если трое из них сидят, ничего не делая. Около 57% (чуть меньше, чем 3/5) переключения контекста были бы напрасны. Несмотря на то, что переключение контекста процессора является невероятно быстрым, оно все равно отнимает время и ресурсы процессора.

Зеленые потоки

Зеленые потоки (green threads) являются примитивным уровнем асинхронного программирования. Зеленый поток — это обычный поток, за исключением того, что переключения между потоками производятся в коде приложения, а не в процессоре. [Gevent](http://www.gevent.org/) — известная Python-библиотека для использования зеленых потоков. Gevent — это зеленые потоки и сетевая библиотека неблокирующего ввода-вывода Eventlet. Gevent.monkey изменяет поведение стандартных библиотек Python таким образом, что они позволяют выполнять неблокирующие операции ввода-вывода. Вот пример использования Gevent для одновременного обращения к нескольким URL-адресам:

import gevent.monkey

from urllib.request import urlopen

gevent.monkey.patch\_all()

urls = ['http://www.google.com', 'http://www.yandex.ru', 'http://www.python.org']

def print\_head(url):

print('Starting {}'.format(url))

data = urlopen(url).read()

print('{}: {} bytes: {}'.format(url, len(data), data))

jobs = [gevent.spawn(print\_head, \_url) for \_url in urls]

gevent.wait(jobs)

Как видите, API-интерфейс Gevent выглядит так же, как и потоки. Однако за кадром он использует сопрограммы (coroutines), а не потоки, и запускает их в цикле событий (event loop) для постановки в очередь. Это значит, что вы получаете преимущества потоков, без понимания сопрограмм, но вы не избавляетесь от проблем, связанных с потоками. Gevent — хорошая библиотека, но только для тех, кто понимает, как работают потоки.

Давайте рассмотрим некоторые аспекты асинхронного программирования. Один из таких аспектов — это цикл событий. Цикл событий — это очередь событий/заданий и цикл, который вытягивает задания из очереди и запускает их. Эти задания называются сопрограммами. Они представляют собой небольшой набор команд, содержащих, помимо прочего, инструкции о том, какие события при необходимости нужно возвращать в очередь.

Функция обратного вызова (callback)

В Python много библиотек для асинхронного программирования, наиболее популярными являются Tornado, Asyncio и Gevent. Давайте посмотрим, как работает [Tornado](http://www.tornadoweb.org/). Он использует стиль обратного вызова (callbacks) для асинхронного сетевого ввода-вывода. Обратный вызов — это функция, которая означает: «Как только это будет сделано, выполните эту функцию». Другими словами, вы звоните в службу поддержки и оставляете свой номер, чтобы они, когда будут доступны, перезвонили, вместо того, чтобы ждать их ответа.  
Давайте посмотрим, как сделать то же самое, что и выше, используя Tornado:

import tornado.ioloop

from tornado.httpclient import AsyncHTTPClient

urls = ['http://www.google.com', 'http://www.yandex.ru', 'http://www.python.org']

def handle\_response(response):

if response.error:

print("Error:", response.error)

else:

url = response.request.url

data = response.body

print('{}: {} bytes: {}'.format(url, len(data), data))

http\_client = AsyncHTTPClient()

for url in urls:

http\_client.fetch(url, handle\_response)

tornado.ioloop.IOLoop.instance().start()

Предпоследняя строка кода вызывает метод AsyncHTTPClient.fetch, который получает данные по URL-адресу неблокирующим способом. Этот метод выполняется и возвращается немедленно. Поскольку каждая следующая строка будет выполнена до того, как будет получен ответ по URL-адресу, невозможно получить объект, как результат выполнения метода. Решение этой проблемы заключается в том, что метод fetch вместо того, чтобы возвращать объект, вызывает функцию с результатом или обратный вызов. Обратный вызов в этом примере — handle\_response.

В примере вы можете заметить, что первая строка функции handle\_response проверяет наличие ошибки. Это необходимо, потому что невозможно обработать исключение. Если исключение было создано, то оно не будет отрабатываться в коде из-за цикла событий. Когда fetch выполняется, он запускает HTTP-запрос, а затем обрабатывает ответ в цикле событий. К тому моменту, когда возникнет ошибка, стек вызовов будет содержать только цикл событий и текущую функцию, при этом нигде в коде не сработает исключение. Таким образом, любые исключения, созданные в функции обратного вызова, прерывают цикл событий и останавливают выполнение программы. Поэтому все ошибки должны быть переданы как объекты, а не обработаны в виде исключений. Это означает, что если вы не проверили наличие ошибок, то они не будут обрабатываться.  
Другая проблема с обратными вызовами заключается в том, что в асинхронном программировании единственный способ избегать блокировок — это обратный вызов. Это может привести к очень длинной цепочке: обратный вызов после обратного вызова после обратного вызова. Поскольку теряется доступ к стеку и переменным, вы в конечном итоге переносите большие объекты во все ваши обратные вызовы, но если вы используете сторонние API-интерфейсы, то не можете передать что-либо в обратный вызов, если он этого не может принять. Это также становится проблемой, потому что каждый обратный вызов действует как поток. Например, вы хотели бы вызвать три API-интерфейса и дождаться, пока все три вернут результат, чтобы его обобщить. В Gevent вы можете это сделать, но не с обратными вызовами. Вам придется немного поколдовать, сохраняя результат в глобальной переменной и проверяя в обратном вызове, является ли результат окончательным.

Сравнения

Если вы хотите предотвратить блокировку ввода-вывода, вы должны использовать либо потоки, либо асинхронность. В Python вы выбираете между зелеными потоками и асинхронным обратным вызовом. Вот некоторые из их особенностей:

Зеленые потоки

* потоки управляются на уровне приложений, а не аппаратно;
* включают в себя все проблемы потокового программирования.

Обратный вызов

* сопрограммы невидимы для программиста;
* обратные вызовы ограничивают использование исключений;
* обратные вызовы трудно отлаживаются.

Как решить эти проблемы?

Вплоть до Python 3.3 зеленые потоки и обратный вызов были оптимальными решениями. Чтобы превзойти эти решения, нужна поддержка на уровне языка. Python должен каким-то образом частично выполнить метод, прекратить выполнение, поддерживая при этом объекты стека и исключения. Если вы знакомы с концепциями Python, то понимаете, что я намекаю на генераторы. Генераторы позволяют функции возвращать список по одному элементу за раз, останавливая выполнение до того момента, когда следующий элемент будет запрошен. Проблема с генераторами заключается в том, что они полностью зависят от функции, вызывающей его. Другими словами, генератор не может вызвать генератор. По крайней мере так было до тех пор, пока в [PEP 380](https://docs.python.org/3/whatsnew/3.3.html#pep-380) не добавили синтаксис yield from, который позволяет генератору получить результат другого генератора. Хоть асинхронность и не является главным назначением генераторов, они содержат весь функционал, чтобы быть достаточно полезными. Генераторы поддерживают стек и могут создавать исключения. Если бы вы написали цикл событий, в котором бы запускались генераторы, у вас получилась бы отличная асинхронная библиотека. Именно так и была создана библиотека [Asyncio](https://docs.python.org/3/library/asyncio.html" \t "_blank).

Все, что вам нужно сделать, это добавить декоратор @coroutine, а Asyncio добавит генератор в сопрограмму. Вот пример того, как обработать те же три URL-адреса, что и раньше:

import asyncio

import aiohttp

urls = ['http://www.google.com', 'http://www.yandex.ru', 'http://www.python.org']

@asyncio.coroutine

def call\_url(url):

print('Starting {}'.format(url))

response = yield from aiohttp.get(url)

data = yield from response.text()

print('{}: {} bytes: {}'.format(url, len(data), data))

return data

futures = [call\_url(url) for url in urls]

loop = asyncio.get\_event\_loop()

loop.run\_until\_complete(asyncio.wait(futures))

**Прим. перев.** В примерах используется aiohttp версии 1.3.5. В последней версии библиотеки [синтаксис другой](https://aiohttp.readthedocs.io/en/stable/client.html#make-a-request).

Несколько особенностей, которые нужно отметить:

* ошибки корректно передаются в стек;
* можно вернуть объект, если необходимо;
* можно запустить все сопрограммы;
* нет обратных вызовов;
* строка 10 не выполнится до тех пор, пока строка 9 не будет полностью выполнена.

Единственная проблема заключается в том, что объект выглядит как генератор, и это может вызвать проблемы, если на самом деле это был генератор.

Async и Await

Библиотека Asyncio довольно мощная, поэтому Python решил сделать ее стандартной библиотекой. В синтаксис также добавили ключевое слово async. Ключевые слова предназначены для более четкого обозначения асинхронного кода. Поэтому теперь методы не путаются с генераторами. Ключевое слово async идет до def, чтобы показать, что метод является асинхронным. Ключевое слово awaitпоказывает, что вы ожидаете завершения сопрограммы. Вот тот же пример, но с ключевыми словами async/await:

import asyncio

import aiohttp

urls = ['http://www.google.com', 'http://www.yandex.ru', 'http://www.python.org']

async def call\_url(url):

print('Starting {}'.format(url))

response = await aiohttp.get(url)

data = await response.text()

print('{}: {} bytes: {}'.format(url, len(data), data))

return data

futures = [call\_url(url) for url in urls]

loop = asyncio.get\_event\_loop()

loop.run\_until\_complete(asyncio.wait(futures))

Программа состоит из метода async. Во время выполнения он возвращает сопрограмму, которая затем находится в ожидании.

Заключение

В Python встроена отличная асинхронная библиотека. Давайте еще раз вспомним проблемы потоков и посмотрим, решены ли они теперь:

* **процессорное переключение контекста**: Asyncio является асинхронным и использует цикл событий. Он позволяет переключать контекст программно;
* **состояние гонки:** поскольку Asyncio запускает только одну сопрограмму и переключается только в точках, которые вы определяете, ваш код не подвержен проблеме гонки потоков;
* **взаимная/активная блокировка:** поскольку теперь нет гонки потоков, то не нужно беспокоиться о блокировках. Хотя взаимная блокировка все еще может возникнуть в ситуации, когда две сопрограммы вызывают друг друга, это настолько маловероятно, что вам придется постараться, чтобы такое случилось;
* **исчерпание ресурсов:** поскольку сопрограммы запускаются в одном потоке и не требуют дополнительной памяти, становится намного сложнее исчерпать ресурсы. Однако в Asyncio есть пул «исполнителей» (executors), который по сути является пулом потоков. Если запускать слишком много процессов в пуле исполнителей, вы все равно можете столкнуться с нехваткой ресурсов.

Несмотря на то, что Asyncio довольно хорош, у него есть и проблемы. Во-первых, Asyncio был добавлен в Python недавно. Есть некоторые недоработки, которые еще не исправлены. Во-вторых, когда вы используете асинхронность, это значит, что весь ваш код должен быть асинхронным. Это связано с тем, что выполнение асинхронных функций может занимать слишком много времени, тем самым блокируя цикл событий.

Существует несколько вариантов асинхронного программирования в Python. Вы можете использовать зеленые потоки, обратные вызовы или сопрограммы. Хотя вариантов много, лучший из них — Asyncio. Если используете Python 3.5, то вам лучше использовать эту библиотеку, так как она встроена в ядро ​​python.