AsyncIO

June 09, 2023 18:33 / asyncio gevent python / 3 comments

assyncio

I'd like to put forth my current thinking about asyncio. I hope this will answer some of the questions I've received as to whether Peewee will one day support asyncio, but moreso I hope it will encourage some readers (especially in the web development crowd) to question whether asyncio is appropriate for their project, and if so, look into alternatives like gevent.

What is it

asyncio is a Python API that works in tandem with language-level support to provide a mechanism for running certain operations (like reading from a socket) concurrently. asyncio uses function-coloring and special async-only hooks for iteration and context managers to accomplish this. The popular premise is if you liberally sprinkle some magic `async` and `await` keywords in your blocking code, you will start going fast.

Why I hate it

I think it's a terrible design. The main problem, in my point-of-view, is that in order to utilize asyncio's power, every layer of your stack must be implemented with asyncio in mind. As a result, popular and mature libraries like psycopg2, pymysql and redis-py either do not work, or must provide a parallel "async-friendly" implementation. redis-py does this for example, and the result is on the order of 5,000 lines of nearly-duplicate code to provide an async-friendly implementation of a Redis client (instead of "def" it's "async def" everywhere).

It takes many years of hard-won experience to make a stable, mature software library. asyncio is incompatible with most of these libraries, and the burden is pushed to the library maintainer to ship an asyncio-friendly implementation. Besides the tremendous waste, this has also given rise to a slew of immature "async-specific" libraries that must handle everything from the protocol-level socket handling, all the way up to the top-level APIs and services. Other libraries just punt their "async" code into the asyncio threadpool executor and hope for the best, or worse. This is consequence of asyncio itself being an inner-platform, that must replicate (at the language level!) all the iteration, control-flow, and structure of the host language while also integrating with standard "blocking" Python.

Another fundamental problem I have with asyncio is that, outside of dubious examples and benchmarks, I don't believe its performance benefits outweigh its complexity for the *vast majority* of applications. Python still has a global interpret lock, and despite recent improvements, it's still a **slow** language. asyncio can only provide performance improvements when your application is performing IO, as such your application must spend more time doing IO than any other type of processing. For a typical web application, a significant percentage of the total time a request takes will be spent in Python (rendering a template, for example) - the amount of time spent reading-from and writing-to the database driver socket, for example, is often negligible. Things like connection pools can further reduce latencies by eliminating the need to set-up and tear-down connections every request. Internet-facing servers can buffer requests and responses, insulating the Python server from slow clients. Because roundtrip latency is of prime concern for web applications, even when slow socket operations are necessary they are usually performed outside the request/response cycle.

Unlike threads, asyncio uses cooperative multi-tasking, which requires explicitly demarcating the boundaries in your code where you want to yield to the event loop. This makes it possible for a small mistake or keyword omission to render the whole asyncio edifice useless. Similarly, even when the boundaries are in place, a small fragment of CPU-intensive code can tie-up the interpreter, preventing the event-loop from receiving control in a timely manner. While process-based parallelism is often seen as a solution to this type of problem, multiprocessing has its own quirks (especially when `fork()` is unavailable) -- and now you've got two separate concurrency mechanisms (coroutines run by the event loop, and a collection of child processes which may or may not be managed by the loop). I suspect that there's a decent bit of code out there that purports to be async but, due to a bug or oversight, does not actually work!

I also believe asyncio suffers from shortcomings in its design, and the result is this complicated monstrosity we have today. For an illustrative example, take a look at the patch to add something as fundamental as context locals. To wield asyncio proficiently, one must understand a very wide array of objects, interfaces and how they interact. Error-handling, cancellation, backpressure and orchestration all require care to implement correctly. It seems to me to be the *least* pythonic way to approach concurrent programming.

The "zen" of Python indicates, among other things:

- Beautiful is better than ugly
- Simple is better than complex
- Complex is better than complicated
- Readability counts (ed: "count" the number of times you use the `async` keyword)
- Errors should never pass silently
- There should be one-- and preferably only one --obvious way to do it.
- If the implementation is hard to explain, it's a bad idea.

My opinion is that asyncio does not meet any of these criteria. If I'm being generous, I will say that it *does* succeed on "explicit is better than implicit", but so would Java - and this is Python we're talking about, a language I personally like for it's ridiculous dynamism and ease-of-use.

Unfortunately, this hasn't stopped a cargo-cult from forming (especially among the JavaScript-adjacent crowd). asyncio web frameworks, orms, database drivers (not even sqlite was spared!!), wsgi servers, and a host more have sprung up across github - replete with twee logos and badge-encrusted READMEs. Whether they're actually going fast or not is anyone's guess. The best you're likely to see is some dubious graph from a half-baked benchmark.

The problem with this in my eyes, is that it promulgates the idea that, if you are building a "modern" web application with Python, you should be using asyncio. There are absolutely valid problems which are best solved with an async solution -- but your CRUD app or short-form video sharing site likely aren't it. Replacing your `pymysql` with `aiomysql` is not going to make you webscale, either. Nevertheless, here we are. There's literally asyncio support in jinja2(?). This feels reminiscent of the early 2010's when, if you wanted to build a modern web application, you needed to be using an eventually-consistent NoSQL database. Unlike the NoSQL fad, asyncio infects *all* aspects of your code.

Alternatives

I've long been a proponent of gevent, as it does not suffer from the issues I've described above (except those fundamental to cooperative multitasking or Python itself). Unlike asyncio, gevent transforms all the blocking calls in your application into non-blocking calls that yield control back to the event loop. As a result, you can continue to use the libraries you're familiar with, and the exact same codebase can be run with or without gevent "enabled". C libraries may need to provide hooks for event-loops - for example psycopg2 does this - but generally everything just works. No need to worry about coroutines, coroutine functions, tasks, futures, loops and policies, `async/await`, etc. Additionally, gevent provides an interface that matches Python's own `threading` module, making it familiar to programmers who've done thread-based concurrency.

An an example, here is a gevent-friendly script that spawns 10 workers, each of which sleeps for 1s (simulating some kind of network IO):

```python
from gevent import monkey; monkey.patch_all()
from gevent.pool import Pool
import time

def work():
    time.sleep(1)  # Or use requests.get() or something else IO-related here.
    print('Done')

start = time.perf_counter()
p = Pool()
for i in range(10):
    p.spawn(work)

p.join()
print(round(time.perf_counter() - start, 3))  # Prints 1.001 or 1.002 for me.
```

gevent eliminates the need to:

- Find and use an async-aware stack of software libraries, or maintain parallel codebases for async vs non-async.
- Learn a complex new set of APIs.
- Use special keywords, implement special object protocols, or worry about function coloring.
- Indicate every place blocking IO occurs.

The bottom-line is that if I were tasked with implementing a server that must handle thousands of concurrent users (or substitute your other favorite async use-case), I would pick gevent or a tool like golang over asyncio every time. Go, for instance, uses an M:N threading model for parallelism, has lightweight threads, provides well-integrated language-level concurrency support, and importantly does not require special purpose-built libraries to benefit from any of these features. Plus it has strong typing, something even the type-hinters should love.

I don't expect that the asyncio excitement will slow down any time soon. If anything I anticipate the opposite as some of the less-pleasant aspects get smoothed-over and libraries become more mature. Nonetheless, hopefully this clarifies my stance on asyncio, and my reasons for preferring gevent when an async solution is wanted.

Threads

I'll end with an unrelated note that regular, boring old `threading.Thread` can often provide very good performance with none of the asyncio drawbacks. You probably can't run 10,000 of them at a time, but they are an **extremely** useful tool and play nicely with existing codebases. Threads also have some benefits when working with C libraries, particularly those that may perform their own IO or CPU-intensive operations, as the C code can release the GIL at important points, providing some parallelism. Additionally, threads are pre-emptively scheduled, so even if you saturate the CPU your tasks will run more-or-less evenly.

As an example, consider two Cython functions:

```
from posix.unistd cimport usleep

def sleep_nogil(n):
    cdef uint32_t dur = n * 1000
    with nogil:
        usleep(dur)

def sleep_gil(n):
    cdef uint32_t dur = n * 1000
    usleep(dur)
```

If I spawn 5 Python threads which call the above functions, what do you imagine the timings would be for them to finish?

```
def run_nogil():
    sleep_nogil(5000)

def run_gil():
    sleep_gil(5000)

threads = [threading.Thread(target=run_nogil) for _ in range(5)]
s = time.perf_counter()
for t in threads: t.start()
for t in threads: t.join()
print('C code releases GIL:', round(time.perf_counter() - s, 3))

threads = [threading.Thread(target=run_gil) for _ in range(5)]
s = time.perf_counter()
for t in threads: t.start()
for t in threads: t.join()
print('C code holds GIL:', round(time.perf_counter() - s, 3))

# C code releases GIL: 5.001
# C code holds GIL: 25.002
```

The first example, which releases the GIL, completes in 5.001 seconds on my machine, while the latter completes (as expected) in 25.002 seconds. This is all to make the point that, even with the GIL, there are always ways to find additional speed in performance-critical code.

A couple excellent posts if you'd like to read more:

- I Don't Understand AsyncIO, from Armin Ronacher.
- Asynchronous Python and Databases, from Mike Bayer.
