Using `asyncio` to facilitate an application server herd

Abstract

`asyncio` is a Python library which provides both high and low level APIs for writing asynchronous I/O code with the ubiquitous `async/await` syntax. Its high-level APIs cover common cases for single-threaded networking, while its low-level APIs allow for greater control over the coroutines being run. Python's simplicity and popularity correlate to powerful standard library modules, and `asyncio` is no exception. Combined with Python's sweeping memory safety, single-threaded networking concurrency is easy.

1 Python as a Language

1.1 Ease of Use

Python is a notoriously simple language. Compared to languages popular for concurrent networking such as JavaScript, C#, and even Go, Python's philosophy of clarity and brevity provides a smooth programming experience (Python being an oft-used language for prototyping, case in point this project), as well as a codebase well-suited for future analysis and augmentation.

Python's choice of syntax for coroutines, in `async/await` , is ubiquitous; it appears in the first two languages mentioned above (JS and C#), as well as in newer languages such as Rust, Kotlin, Dart, and Nim. While coroutines started in theory in the 1960s, the `async/await` syntax is only a decade old - yet it has taken over mainstream concurrent programming. Other languages with notable facilities for concurrent programming, such as Go and Erlang, typically implement some form of lightweight multi-threading (see: 'goroutines' and 'Erlang processes'), as opposed to coroutines as seen in Python and JavaScript; in all languages with coroutines, some form of the `async/await` paradigm is used - even in C++20, which uses keywords like `co_await` and `future<T>` to signify when functions are executed asynchronously.

1.2 Compatibility

`asyncio` is a relatively new library (having been added to the language in 2014) and as such frequently experiences changes to its API. The largest example of an update in API is the addition of `asyncio.run()` , which occured in Python 3.7, and was made stable in Python 3.8. The documentation for Python's 3.8 update itself acknowledges the breaking change - when comparing the new function to old methods of starting the event loop, it remarks that:

> The actual implementation is significantly more complex. Thus, asyncio.run() should be the preferred way of running asyncio programs.

While by now, the `asyncio` library is at least 7 years old, it is worth considering the problems of future maintenance and updates to the codebase. There are plenty of slight changes and deprecations to parts of the library here and there; this may put engineers in consistent jobs as codebase janitors.

1.3 Clarity and Semantics

Python suffers from a problem with explicitness. It is dynamically typed, so the code loses not only an awareness of the types of arguments being serialized/de-serialized, encoded/decoded, but also built-in documentation on the flow of execution through type annotation. Nonetheless, the tradeoff for loss of information in exchange for improvement in development agility should be considered when discussing the lack of a type system.

This is also a problem of exception handling itself - a programmer cannot know what sort of exceptions a function throws until they catch one themselves. The documentation for the library is sparse when it comes to describing all the possible ways things might go wrong. When it comes to testing all the possible ways the program might err or fail, Python may seem more hands on and empirical. Many other languages come with more detailed/complete error description elements; e.g. Rust and Haskell.

1.4 Memory-management, threading

Memory-management should not be a problem in Python - the language is entirely memory-safe. The same cannot be said for some lower-level competitors (C++ and unsafe Rust), but it is a tradeoff of performance for safety. In typical memory safety, Python and Java are roughly equivalent - both languages use garbage collection and use exception handling. Python's null value is perhaps safer than Java's, and the lack of pointers/references makes Python less powerful but certainly safer. It wouldn't be inaccurate to claim that Python's cautious safety is superior to that of most competing languages - the cost is performance and control.

Multithreading in Python is done not for performance, but for concurrent I/O. The documentation for the standard library's `threading` states:

> In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once.... threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

The whole purpose of `asyncio` is to provide single-threaded concurrent I/O, and as such, no threads are used or need to be used in the implementation of a web-server (the prototype certainly does not need them). This design reflects that of Node.js', the primary counterpart to `asyncio` .

2 Working with `asyncio`

2.1 API

The `asyncio` high-level API is exceedingly simple and high-level. There are two main functions: `open_connection` is for establishing a network connection as a client, and `start_server` is for setting up a socket server. These two functions provide all the functionality needed for most cases, and certainly for the app server prototype we designed.

There is also of course a low-level API for programmers who need it, which should raise confidence that the library carries enough facilities to implement a large web app stack.

Being a new API, there are enough oddities that might hinder the speed of development under `asyncio` . For example, if you want to run a set of coroutines concurrently, the documentation suggests creating `Task` s out of them. To run multiple `Task` s, the documentation suggests using `asyncio.gather()` . However, the function does not take an iterable, but a variadic sequence of arguments - so, if you want to run n many of these tasks concurrently, with n variable during runtime (in an iterable), you have to know what the 'splat' operator is - a mild inconvenience for what should be a common use case of passing a series of coroutines to a function designed to run multiple coroutines at the same time.

Nonetheless, `asyncio` is easy to write and maintain - its syntax resembles synchronous Python, with only the addition of two keywords to acknowledge the event loop. One possible drawback is that everything in the `asyncio` environment must also be made asynchronous, but with such a simple API, extending programs with coroutines galore should not be problematic.

2.2 Latest features

For `asyncio` , it is absolutely preferable to write code using the newest features of Python. If the code is being written from scratch, obviously the latest (if not pre-release) stable API should be used, to help defend against slow deprecation. If we had an old codebase already written, the `asyncio` API does support the same operations it promoted as first-class options in earlier versions of the library, with minor changes, so that it shouldn't be too problematic to use older versions. The server prototype in fact uses very few elements from the most recent version of the library - nonetheless, `asyncio.run()` may prove very convenient when writing a server without needing to understand the implemntation underneath.

3 Discussing Performance

3.1 Python is not the fastest

Python is undeniably a slow language. If speed is a powerful concern, using compiled languages like Go, C#, or Rust are all better options. For example, if the data sent back and forth requires a lot of internal processing, the servers might be caught up in CPU-bound operations rather than I/O-bound operations, which defeats the purpose of using the library (i.e. that I/O is the primary bottleneck); Node.js suffers from the same problem. There are C-based extensions to the language, as well as JIT compilers (PyPy comes to mind) - however, these are not practical to extend and are often incompatible with recent versions of `asyncio` . Regardless, it isn't expected that language performance is a huge issue for the application discussed.

3.2 `asyncio` and co. are not the fastest

It would be hard to describe `asyncio` itself as the fastest interface for asynchronous networking available to Python, and especially not in comparison with other languages' libraries. Near the time of its introduction, `aiohttp` notably was bottlenecked by a slow HTTP parser - which eventually was replaced with a significantly faster parser written in C. For performance improvements, extensions related to C are common.

`uvloop`

There also exists the library `uvloop` , which replaces the `asyncio` built-in event loop with a reputedly faster event loop. Its benchmarks boast TCP server performance comparable to Golang's, almost twice that of Node.js's performance. While its benchmarks are dated by now (2016), it may be worth considering - especially as it implements the same interface as the `asyncio` event loop, allowing for a seamless transition between the two implementations. In effect, we would be using `asyncio` , but with a free performance bonus. One caveat is that the `uvloop` library only works on Unix machines - but this shouldn't be a problem as servers generally run distributions of Linux.

3.3 Synchronous, multithreaded?

The design choice of stackful coroutines in a single-threaded event loop over a multithreaded one is ultimately better. If we are to use Python, multithreaded performance suffers, thanks to the Global Interpreter Lock. In other languages which better support threading for performance, there is still a massive consumption of resources for having to expend a new thread's stack for each request.

Asynchronous programming is less wasteful and, when written properly, can provide a tighter flow of execution, without costing as much memory/OS involvement.

4 Conclusion

The change in architecture with `asyncio` as the driving force is a good decision. The library provides effortless asynchronous networking capabilities, in all of conception, prototyping, development, and maintenance - this is due to Python's inherent simplicity and brevity. While some extensions or adjustments to the environment may be desirable for the sake of performance, ultimately the `asyncio` library is a fine choice for our application.