I don't understand Python's Asyncio

written on Sunday, October 30, 2016

Recently I started looking into Python's new asyncio module a bit more. The reason for this is that I needed to do something that works better with evented IO and I figured I might give the new hot thing in the Python world a try. Primarily what I learned from this exercise is that I it's a much more complex system than I expected and I am now at the point where I am very confident that I do not know how to use it properly.

It's not conceptionally hard to understand and borrows a lot from Twisted, but it has so many elements that play into it that I'm not sure any more how the individual bits and pieces are supposed to go together. Since I'm not clever enough to actually propose anything better I just figured I share my thoughts about what confuses me instead so that others might be able to use that in some capacity to understand it.

The Primitives

asyncio is supposed to implement asynchronous IO with the help of coroutines. Originally implemented as a library around the *yield* and *yield from* expressions it's now a much more complex beast as the language evolved at the same time. So here is the current set of things that you need to know exist:

- event loops
- event loop policies
- awaitables
- coroutine functions
- old style coroutine functions
- coroutines
- coroutine wrappers
- generators
- futures
- concurrent futures
- tasks
- handles
- executors
- transports
- protocols

In addition the language gained a few special methods that are new:

- `__aenter__` and `__aexit__` for asynchronous *with* blocks
- `__aiter__` and `__anext__` for asynchronous iterators (async loops and async comprehensions). For extra fun that protocol already changed once. In 3.5 it returns an awaitable (a coroutine) in Python 3.6 it will return a newfangled async generator.
- `__await__` for custom awaitables

That's quite a bit to know and the documentation covers those parts. However here are some notes I made on some of those things to understand them better:

Event Loops

The event loop in asyncio is a bit different than you would expect from first look. On the surface it looks like each thread has one event loop but that's not really how it works. Here is how I think this works:

- if you are the main thread an event loop is created when you call `asyncio.get_event_loop()`
- if you are any other thread, a runtime error is raised from `asyncio.get_event_loop()`
- You can at any point `asyncio.set_event_loop()` to bind an event loop with the current thread. Such an event loop can be created with the `asyncio.new_event_loop()` function.
- Event loops can be used without being bound to the current thread.
- `asyncio.get_event_loop()` returns the thread bound event loop, it does not return the currently running event loop.

The combination of these behaviors is super confusing for a few reasons. First of all you need to know that these functions are delegates to the underlying event loop policy which is globally set. The default is to bind the event loop to the thread. Alternatively one could in theory bind the event loop to a greenlet or something similar if one would so desire. However it's important to know that library code does not control the policy and as such cannot reason that asyncio will scope to a thread.

Secondly asyncio does not require event loops to be bound to the context through the policy. An event loop can work just fine in isolation. However this is the first problem for library code as a coroutine or something similar does not know which event loop is responsible for scheduling it. This means that if you call `asyncio.get_event_loop()` from within a coroutine you might not get the event loop back that ran you. This is also the reason why all APIs take an optional explicit loop parameter. So for instance to figure out which coroutine is currently running one cannot invoke something like this:

```
def get_task():
    loop = asyncio.get_event_loop()
    try:
        return asyncio.Task.get_current(loop)
    except RuntimeError:
        return None
```

Instead the loop has to be passed explicitly. This furthermore requires you to pass through the loop explicitly everywhere in library code or very strange things will happen. Not sure what the thinking for that design is but if this is not being fixed (that for instance `get_event_loop()` returns the actually running loop) then the only other change that makes sense is to explicitly disallow explicit loop passing and require it to be bound to the current context (thread etc.).

Since the event loop policy does not provide an identifier for the current context it also is impossible for a library to "key" to the current context in any way. There are also no callbacks that would permit to hook the tearing down of such a context which further limits what can be done realistically.

Awaitables and Coroutines

In my humble opinion the biggest design mistake of Python was to overload iterators so much. They are now being used not just for iteration but also for various types of coroutines. One of the biggest design mistakes of iterators in Python is that *StopIteration* bubbles if not caught. This can cause very frustrating problems where an exception somewhere can cause a generator or coroutine elsewhere to abort. This is a long running issue that Jinja for instance has to fight with. The template engine internally renders into a generator and when a template for some reason raises a *StopIteration* the rendering just ends there.

Python is slowly learning the lesson of overloading this system more. First of all in 3.something the asyncio module landed and did not have language support. So it was decorators and generators all the way down. To implemented the *yield from* support and more, the *StopIteration* was overloaded once more. This lead to surprising behavior like this:

```
>>> def foo(n):
...  if n in (0, 1):
...   return [1]
...  for item in range(n):
...   yield item * 2
...
>>> list(foo(0))
[]
>>> list(foo(1))
[]
>>> list(foo(2))
[0, 2]
```

No error, no warning. Just not the behavior you expect. This is because a *return* with a value from a function that is a generator actually raises a *StopIteration* with a single arg that is not picked up by the iterator protocol but just handled in the coroutine code.

With 3.5 and 3.6 a lot changed because now in addition to generators we have coroutine objects. Instead of making a coroutine by wrapping a generator there is no a separate object which creates a coroutine directly. It's implemented by prefixing a function with `async`. For instance `async def x()` will make such a coroutine. Now in 3.6 there will be separate async generators that will raise *AsyncStopIteration* to keep it apart. Additionally with Python 3.5 and later there is now a future import (`generator_stop`) that will raise a *RuntimeError* if code raises *StopIteration* in an iteration step.

Why am I mentioning all this? Because the old stuff does not really go away. Generators still have *send* and *throw* and coroutines still largely behave like generators. That is a lot of stuff you need to know now for quite some time going forward.

To unify a lot of this duplication we have a few more concepts in Python now:

- awaitable: an object with an `__await__` method. This is for instance implemented by native coroutines and old style coroutines and some others.
- coroutinefunction: a function that returns a native coroutine. Not to be confused with a function returning a coroutine.
- a coroutine: a native coroutine. Note that old asyncio coroutines are not considered coroutines by the current documentation as far as I can tell. At the very least `inspect.iscoroutine` does not consider that a coroutine. It's however picked up by the future/awaitable branches.

In particularly confusing is that `asyncio.iscoroutinefunction` and `inspect.iscoroutinefunction` are doing different things. Same with `inspect.iscoroutine` and `inspect.iscoroutinefunction`. Note that even though inspect does not know anything about asycnio legacy coroutine functions in the type check, it is apparently aware of them when you check for awaitable status even though it does not conform to `__await__`.

Coroutine Wrappers

Whenever you run `async def` Python invokes a thread local coroutine wrapper. It's set with `sys.set_coroutine_wrapper` and it's a function that can wrap this. Looks a bit like this:

```
>>> import sys
>>> sys.set_coroutine_wrapper(lambda x: 42)
>>> async def foo():
...  pass
...
>>> foo()
__main__:1: RuntimeWarning: coroutine 'foo' was never awaited
42
```

In this case I never actually invoke the original function and just give you a hint of what this can do. As far as I can tell this is always thread local so if you swap out the event loop policy you need to figure out separately how to make this coroutine wrapper sync up with the same context if that's something you want to do. New threads spawned will not inherit that flag from the parent thread.

This is not to be confused with the asyncio coroutine wrapping code.

Awaitables and Futures

Some things are awaitables. As far as I can see the following things are considered awaitable:

- native coroutines
- generators that have the fake `CO_ITERABLE_COROUTINE` flag set (we will cover that)
- objects with an `__await__` method

Essentially these are all objects with an `__await__` method except that the generators don't for legacy reasons. Where does the `CO_ITERABLE_COROUTINE` flag come from? It comes from a coroutine wrapper (now to be confused with `sys.set_coroutine_wrapper`) that is `@asyncio.coroutine`. That through some indirection will wrap the generator with `types.coroutine` (to to be confused with `types.CoroutineType` or `asyncio.coroutine`) which will re-create the internal code object with the additional flag `CO_ITERABLE_COROUTINE`.

So now that we know what those things are, what are futures? First we need to clear up one thing: there are actually two (completely incompatible) types of futures in Python 3. `asyncio.futures.Future` and `concurrent.futures.Future`. One came before the other but they are also also both still used even within asyncio. For instance `asyncio.run_coroutine_threadsafe()` will dispatch a coroutine to a event loop running in another thread but it will then return a `concurrent.futures.Future` object instead of a `asyncio.futures.Future` object. This makes sense because only the `concurrent.futures.Future` object is thread safe.

So now that we know there are two incompatible futures we should clarify what futures are in asyncio. Honestly I'm not entirely sure where the differences are but I'm going to call this "eventual" for the moment. It's an object that eventually will hold a value and you can do some handling with that eventual result while it's still computing. Some variations of this are called deferreds, others are called promises. What the exact difference is is above my head.

What can you do with a future? You can attach a callback that will be invoked once it's ready or you can attach a callback that will be invoked if the future fails. Additionally you can `await` it (it implements `__await__` and is thus awaitable). Additionally futures can be cancelled.

So how do you get such a future? By calling `asyncio.ensure_future` on an awaitable object. This will also make a good old generator into such a future. However if you read the docs you will read that `asyncio.ensure_future` actually returns a `Task`. So what's a task?

Tasks

A task is a future that is wrapping a coroutine in particular. It works like a future but it also has some extra methods to extract the current stack of the contained coroutine. We already saw the tasks mentioned earlier because it's the main way to figure out what an event loop is currently doing via `Task.get_current`.

There is also a difference in how cancellation works for tasks and futures but that's beyond the scope of this. Cancellation is its own entire beast. If you are in a coroutine and you know you are currently running you can get your own task through `Task.get_current` as mentioned but this requires knowledge of what event loop you are dispatched on which might or might not be the thread bound one.

It's not possible for a coroutine to know which loop goes with it. Also the *Task* does not provide that information through a public API. However if you did manage to get hold of a task you can currently access `task._loop` to find back to the event loop.

Handles

In addition to all of this there are handles. Handles are opaque objects of pending executions that cannot be awaited but they can be cancelled. In particular if you schedule the execution of a call with `call_soon` or `call_soon_threadsafe` (and some others) you get that handle you can then use to cancel the execution as a best effort attempt but you can't wait for the call to actually take place.

Executors

Since you can have multiple event loops but it's not obvious what the use of more than one of those things per thread is the obvious assumption can be made that a common setup is to have N threads with an event loop each. So how do you inform another event loop about doing some work? You cannot schedule a callback into an event loop in another thread *and* get the result back. For that you need to use executors instead.

Executors come from `concurrent.futures` for instance and they allow you to schedule work into threads that itself is not evented. For instance if you use `run_in_executor` on the event loop to schedule a function to be called in another thread. The result is then returned as an asyncio coroutine instead of a concurrent coroutine like `run_coroutine_threadsafe` would do. I did not yet have enough mental capacity to figure out why those APIs exist, how you are supposed to use and when which one. The documentation suggests that the executor stuff could be used to build multiprocess things.

Transports and Protocols

I always though those would be the confusing things but that's basically a verbatim copy of the same concepts in Twisted. So read those docs if you want to understand them.

How to use asyncio

Now that we know roughly understand asyncio I found a few patterns that people seem to use when they write asyncio code:

- pass the event loop to all coroutines. That appears to be what a part of the community is doing. Giving a coroutine knowledge about what loop is going to schedule it makes it possible for the coroutine to learn about its task.
- alternatively you require that the loop is bound to the thread. That also lets a coroutine learn about that. Ideally support both. Sadly the community is already torn of what to do.
- If you want to use contextual data (think thread locals) you are a bit out of luck currently. The most popular workaround is apparently atlassian's `aiolocals` which basically requires you to manually propagate contextual information into coroutines spawned since the interpreter does not provide support for this. This means that if you have a utility library spawning coroutines you will lose context.
- Ignore that the old coroutine stuff in Python exists. Use 3.5 only with the new `async def` keyword and co. In particular you will need that anyways to somewhat enjoy the experience because with older versions you do not have async context managers which turn out to be very necessary for resource management.
- Learn to restart the event loop for cleanup. This is something that took me longer to realize than I wish it did but the sanest way to deal with cleanup logic that is written in async code is to restart the event loop a few times until nothing pending is left. Since sadly there is no common pattern to deal with this you will end up with some ugly workaround at time. For instance *aiohttp*'s web support also does this pattern so if you want to combine two cleanup logics you will probably have to reimplement the utility helper that it provides since that helper completely tears down the loop when it's done. This is also not the first library I saw do this :(
- Working with subprocesses is non obvious. You need to have an event loop running in the main thread which I suppose is listening in on signal events and then dispatches it to other event loops. This requires that the loop is notified via `asyncio.get_child_watcher().attach_loop(...)`.

- Writing code that supports both async and sync is somewhat of a lost cause. It also gets dangerous quickly when you start being clever and try to support `with` and `async with` on the same object for instance.
- If you want to give a coroutine a better name to figure out why it was not being awaited, setting `__name__` doesn't help. You need to set `__qualname__` instead which is what the error message printer uses.
- Sometimes internal type conversations can screw you over. In particular the `asyncio.wait()` function will make sure all things passed are futures which means that if you pass coroutines instead you will have a hard time finding out if your coroutine finished or is pending since the input objects no longer match the output objects. In that case the only real sane thing to do is to ensure that everything is a future upfront.

Context Data

Aside from the insane complexity and lack of understanding on my part of how to best write APIs for it my biggest issue is the complete lack of consideration for context local data. This is something that the node community learned by now. `continuation-local-storage` exists but has been accepted as implemented too late. Continuation local storage and similar concepts are regularly used to enforce security policies in a concurrent environment and corruption of that information can cause severe security issues.

The fact that Python does not even have any store at all for this is more than disappointing. I was looking into this in particular because I'm investigating how to best support Sentry's breadcrumbs for asyncio and I do not see a sane way to do it. There is no concept of context in asyncio, there is no way to figure out which event loop you are working with from generic code and without monkeypatching the world this information will not be available.

Node is currently going through the process of finding a long term solution for this problem. That this is not something to be left ignored can be seen by this being a recurring issue in all ecosystems. It comes up with JavaScript, Python and the .NET environment. The problem is named async context propagation and solutions go by many names. In Go the context package needs to be used and explicitly passed to all goroutines (not a perfect solution but at least one). .NET has the best solution in the form of local call contexts. It can be a thread context, an web request context, or something similar but it's automatically propagating unless suppressed. This is the gold standard of what to aim for. Microsoft had this solved since more than 15 years now I believe.

I don't know if the ecosystem is still young enough that logical call contexts can be added but now might still be the time.

Personal Thoughts

Man that thing is complex and it keeps getting more complex. I do not have the mental capacity to casually work with asyncio. It requires constantly updating the knowledge with all language changes and it has tremendously complicated the language. It's impressive that an ecosystem is evolving around it but I can't help but get the impression that it will take quite a few more years for it to become a particularly enjoyable and stable development experience.

What landed in 3.5 (the actual new coroutine objects) is great. In particular with the changes that will come up there is a sensible base that I wish would have been in earlier versions. The entire mess with overloading generators to be coroutines was a mistake in my mind. With regards to what's in asyncio I'm not sure of anything. It's an incredibly complex thing and super messy internally. It's hard to comprehend how it works in all details. When you can pass a generator, when it has to be a real coroutine, what futures are, what tasks are, how the loop works and that did not even come to the actual IO part.

The worst part is that asyncio is not even particularly fast. David Beazley's live demo hacked up asyncio replacement is twice as fast as it. There is an enormous amount of complexity that's hard to understand and reason about and then it fails on it's main promise. I'm not sure what to think about it but I know at least that I don't understand asyncio enough to feel confident about giving people advice about how to structure code for it.

This entry was tagged asyncio and python