njs blog

- Atom

- My homepage
- Blog archive

Some thoughts on asynchronous API design in a post-async/await world

Sat 05 November 2016

I've recently been exploring the exciting new world of asynchronous I/O libraries in Python 3 – specifically asyncio and curio. These two libraries make some different design choices. This is an essay that I wrote to try to explain to myself what those differences are and why I think they matter, and distill some principles for designing event loop APIs and asynchronous libraries in Python. This is a quickly changing area and the ideas here are very much still under development, so this text probably assumes all kinds of background knowledge and possibly that you live inside my head – but maybe you'll find it interesting anyway. I'd love to hear what you think or discuss further.

Contents:

The curious effectiveness of curio

So here's the tentative conclusion that spurred this essay, and which surprised the heck out of me: the more I work with curio, the more plausible it seems that in a few years, asyncio might find itself relegated to becoming one of those stdlib libraries that savvy developers avoid, like urllib2.

I'm not saying that the library we'll all be using instead will necessarily *be* curio, or that asyncio can't possibly find some way to adapt and avoid this fate, or that you should go switch to curio right now – the practicalities of choosing a library are complicated. Let me put it in bold: **This is not an essay about curio versus asyncio and which one is "the best".** I'll talk a lot about those two libraries, but for present puroposes I'm profoundly uninterested in things like which one wins at such-and-such microbenchmark as of which-ever latest release, and I don't have any personal investment in either. The reason I talk about them is because they make good *illustrative*

examples of two very different design strategies.

The goal of this essay is to understand is the trade-offs between the "curio-style" design strategy versus the "asyncio-style" design strategy. So first, I'll try to articulate a conceptual framework for understanding what these two strategies actually are, and how they differ – this is something I haven't seen discussed elsewhere. Then to make that more concrete, I'll walk through some concrete examples using the two libraries, and see how these underlying design decisions play out in specific real world use cases. It turns out that in these examples, the "curio-style" produces better results; I'll try to pull out the general principles that explain why this happens, and that might give us hints on how to design or improve new APIs for both event loops and for the libraries that use them. Unfortunately, one of the conclusions I come to is that it's hard to see how these advantages could be "retrofitted" to asyncio – but I could be wrong, and at least once we understand them we can have a conversation about how to make Python's async I/O ecosystem as awesome as possible, whatever that ends up looking like; I'll conclude by sketching out some possible directions this could go.

Callback soup considered harmful

The basic difference between asyncio and curio comes down to their attitude towards Python 3.5's new async/await syntax. But before we talk about the best way to use async/await, lets digress to talk about why async/await even matters. ...Actually I'm going to digress even more then that. Let's start by talking about what programming languages are for.

It's easy to forget sometimes just how much work a modern language like Python does to guide and constrain how you put together a program. Like, just for the most basic example, consider how simply juxtaposing two statements `f(); g()` expresses ordering: you know that `g` won't start executing until `f` has finished. Another example – the call stack tracks relationships between callers and callees, allowing us to decompose our program into loosely-coupled subroutines: a function doesn't need to keep track of who called it, it can just say `return` to fire a value into the void, and the language runtime makes some magic happen so the value and the control flow are delivered simultaneously to the right place. Exception handling defines a systematic, structured way to unwind these decoupled subroutines when something goes wrong; if you think about it this trick of taking the call stack and *reusing* it as the unwind stack is really quite clever. `with` blocks build on that by giving us an ergonomic way to pin the lifetime of resources – file handles and so forth – to the dynamic lifetime of these call stacks. Iterators track the state needed to bind control flow to the shape of data.

These tools are so fundamental that they disappear into the background – when we're typing out code we don't usually stop to think about all the wacky ways that things could be done differently. Yet these all had to be invented. In functional languages, `f(); g()` doesn't express ordering. Back in the 1970s, the idea of *limiting* yourself to using just function calls, loops, and if statements – `goto`'s hamstrung siblings – was *incredibly controversial*. There are great living languages that disagree with Python about lots of the points above. But Python's particular toolkit has been refined over decades, and fits together to provide a powerful scaffolding for structuring our code.

...until you want to do some asynchronous I/O. The traditional way to do this is to have an *event loop* and use *callback-based programming*: the event loop keeps a big table of future events that you want to respond to when they happen (e.g., "this socket became readable", "that timer expired"), and for each event you have a corresponding callback. The event loop takes care of checking for events and invoking callbacks, but if you want structure beyond that – like the kind of things we just discussed above: causal sequencing, delegation to and return from subroutines, error unwinding, resource cleanup, iteration – then you get to build that yourself. You can do it, just like you can use `goto` to build loops and function calls. Frameworks like Twisted and its descendents have invented all kinds of useful strategies for keeping these callbacks organized, like protocols and deferreds / *futures <https://docs.python.org/3/library/asyncio-task.html#asyncio.Future>* and even some kind of exception handling – but these are still a pretty thin layer of structure on top of the underlying unstructured callback soup, and from the perspective of regular Python they're like some other mutant alternative programming language.

That's why PEP 492 and async/await are so exciting: they let us take Python's regular toolkit for solving these problems, and start using it in asynchronous code. Which is awesome, because frankly, Twisted, I love you and deferreds are pretty cool, but as abstract languages for describing computation go then real-actual-Python is wayyy better.

And with that background, then, I think I can articulate the key difference between asyncio-style event-loop APIs and curio-style event-loop APIs:

Asyncio is first and foremost a traditional callback-based API, with async/await layered on top as a supplementary tool. And if you're starting from a callback-oriented base, then this is a great addition: async/await provide a major boost in usability without disrupting the basic framework. Asyncio is what we might call a "hybrid" system: callbacks *plus* async/await.

Curio takes this a step further, and throws out the callback API altogether; it's async/await all the way down. Specifically, it still has an event loop, but instead of managing arbitrary callbacks, it manages async functions; there's exactly one way it can respond when an event fires, and that's by resuming an async call-stack. I'll call this the "async/await-native" approach.

The main point I want to argue in this essay – the point of all the examples below – is that if you're using a hybrid API like asyncio, then you *can* ignore the callback API and write structured async/await code. But, even if you stick

to async/await everywhere, the underlying abstractions are leaky, so don't get the full advantages. Your async/await functions are dumplings of local structure floating on top of callback soup, and this has far-reaching implications for the simplicity and correctness of your code. Python's structuring tools were designed to fit together as a system – e.g., exception handling relies on the call stack, and `with` blocks rely on exception handling – and if you have a mix of structured and unstructured parts, then this creates lots of unnecessary problems, *even if* you stick to the structured async/await layer of the library. In a curio-style async/await-native API, by contrast, your whole program uses this one consistent set of structuring principles, and this consistency – it turns out – has pervasive benefits.

What I'm arguing, in effect, is that asyncio is a victim of its own success: when it was designed, it used the best approach possible; but since then, work inspired by asyncio – like the addition of async/await – has shifted the landscape so that we can do even better, and now asyncio is hamstrung by its earlier commitments.

To make that more specific, let's look at some concrete examples.

Example: a simple proxy server

For our main example, I'll take a simple proxy server, equivalent to `socat -u TCP-LISTEN:$LOCAL_PORT TCP:$REMOTE_HOST:$REMOTE_PORT`. Specifically, given a local port, a remote host, and a remote port, we want to:

1. Listen for connections on the local port.
2. Accept a single connection.
3. Make a connection to the remote host + port.
4. Copy data from the local port to the remote port. (One way only, to keep things simple.)
5. Exit after all the data has been copied.

In addition, I'll follow these rules, to the best of my ability:

- Readability counts: I'll write each version in as elegant a manner as I can.
- No cheating: Since this is a toy one-off program, there are things we could get away with that wouldn't fly if this were real, reusable library code – like using global variables, or leaving open sockets dangling to be cleaned up by the kernel when our program exits. To make this more representative of real code, I'll hold myself to those higher standards.

Three examples

Example #1: asyncio, with callbacks

Let's start by showing what this looks like using a traditional callback approach. (The two examples after this will demonstrate curio and asyncio's version of async/await-based APIs, which is what most people will want to use – this first example is to provide context for those.) I'll demonstrate with asyncio's "protocol" API, though the basic design here is inherited almost directly from the immensely influential Twisted (a Twisted version is also available for the curious). Here's the complete code, and then I'll give some commentary on how it works:

Download source: asyncio-proxy-protocols.py

```
 1 import sys
 2 from functools import partial
 3 import traceback
 4 import asyncio
 5
 6 class OneWayProxySource(asyncio.Protocol):
 7     def __init__(self, loop, server_task_container, dest_host, dest_port):
 8         self.loop = loop
 9         self.server_task = server_task_container[0]
10         self.dest_host = dest_host
11         self.dest_port = dest_port
12
13     def connection_made(self, transport):
14         # Stop listening for new connections
15         self.server_task.cancel()
16
17         # Save our transport
18         self.transport = transport
19
20         # Disable reading until the destination is ready.
21         self.transport.pause_reading()
22
23         # Connect to the destination
24         self.dest_protocol = OneWayProxyDest(self.loop, self.transport)
25         coro = self.loop.create_connection(lambda: self.dest_protocol,
26                                            self.dest_host, self.dest_port)
27         task = self.loop.create_task(coro)
28         def connection_check_for_failure(fut):
29             exc = fut.exception()
30             if exc is not None:
31                 print("Failed to connect:")
32                 # This isn't really right -- it doesn't handle exception
33                 # chaining etc. I lack the will to worry about it.
34                 traceback.print_tb(exc.__traceback__)
```

```
                self.transport.abort()
                self.loop.stop()
        task.add_done_callback(connection_check_for_failure)

    def data_received(self, data):
        self.dest_protocol.send_data(data)

    def connection_lost(self, exc):
        self.dest_protocol.close()

class OneWayProxyDest(asyncio.Protocol):
    def __init__(self, loop, source_transport):
        self.loop = loop
        self.source_transport = source_transport

    def connection_made(self, transport):
        self.transport = transport
        # Okay, now we're ready for data to start flowing
        self.source_transport.resume_reading()

    def send_data(self, data):
        self.transport.write(data)

    def close(self):
        self.transport.write_eof()

    def connection_lost(self, exc):
        self.source_transport.abort()
        self.loop.stop()

def main(source_port, dest_host, dest_port):
    loop = asyncio.get_event_loop()
    server_task_container = []
    coro = loop.create_server(
        lambda: OneWayProxySource(loop, server_task_container,
                                  dest_host, dest_port),

```