What is Zig's "Colorblind" Async/Await?

June 21, 2020 · 11 min read · by **Loris Cro**

An introduction to Zig's curious concurrent programming API.

Zig is a new, general-purpose programming language that is rethinking from the ground up the way languages and related tooling should work. I've already discussed compile-time code execution, and now I'm going to introduce another innovative concept from the language: async/await.

But wait a minute, isn't async/await present also in other languages?

Well yes, but async/await in Zig combines with compile-time execution, allowing functions to implicitly become async and whole libraries to work transparently in both blocking and evented I/O mode in a way that's very much unique to Zig.

Let's see what that means.

Async/Await notation in Zig

While Zig is very innovative, it tries to be a small and simple language. Zig takes a lot of inspiration from the simplicity of C and reserves metaprogramming acrobatics for when you really need them. In general, you should be able to become productive with Zig in a weekend. So, while most Zig features will immediately strike you as pleasant and somewhat familiar, async/await is one example of the familiarity-rule being broken, but for good reason.

A bit of context

When it comes to making use of evented I/O, you tend to have two options in imperative programming languages: callbacks or async/await.

The first case has the advantage of not requiring any addition to the language, but the downside is that now everything has to be based on callbacks and nested closures. The second case basically leaves it to the compiler to break down your function into different "stages", making the whole translation transparent to the user (i.e. to you it still looks like normal, sequential, imperative code) but, unfortunately, it has the side effect of introducing function coloring.

Function coloring

The idea of function coloring is nicely explained by this blog post, but to sum it up: since you can't call async functions from non-async code, you end up with a lot of duplicated effort, where you need to reimplement parts of your standard library and all networking-related libraries to account for async/await. One example of this is Python, where the introduction of async/await in Python 3 birthed projects like aio-libs, whose goal is to reimplement popular networking libraries on top of AsyncIO.

Zig's "colorblind" async/await

Let's walk through how async/await works in Zig by looking at a few examples. To see the following code snippets run, launch Netcat in another terminal in listen mode and, if everything works, you should see "Hello World!" being printed after every run.

Here's the complete Netcat command:

SHELL

```
➜ nc -lk 7000
```

Basics

The first example is a simple blocking program that writes to a socket. Nothing surprising there.

```zig
const net = @import("std").net;

pub fn main() !void {
    const addr = try net.Address.parseIp("127.0.0.1", 7000);
    try send_message(addr);
}

fn send_message(addr: net.Address) !void {
    var socket = try net.tcpConnectToAddress(addr);
    defer socket.close();

    _ = try socket.write("Hello World!\n");
}
```

Adding one special declaration in the source file enables evented I/O.

```zig
const net = @import("std").net;

pub const io_mode = .evented;

pub fn main() !void {
    const addr = try net.Address.parseIp("127.0.0.1", 7000);
    try send_message(addr);
}

fn send_message(addr: net.Address) !void {
    var socket = try net.tcpConnectToAddress(addr);
    defer socket.close();

    _ = try socket.write("Hello World!\n");
}
```

That declaration caused a few changes in the background, one of which is opening the socket in non-blocking mode. This causes the function to become async, but as you can see the way it gets called did not change: it still looks like a normal function invocation, even though it's not.

The Zig code above is functionally equivalent to the following Python code:

```python
import asyncio

async def main():
    await send_message("127.0.0.1", 7000)

async def send_message(addr, port):
    _, writer = await asyncio.open_connection(addr, port)
    writer.write(b"Hello World!\n")
    writer.close()

asyncio.run(main())
```

Expressing concurrency

You've seen now that in Zig there is no extra keyword requirement to launch a coroutine and immediately wait for its completion. So how do you start a coroutine and `await` it afterward then? By using `async` of course!

```zig
const net = @import("std").net;

pub const io_mode = .evented;

pub fn main() !void {
    const addr = try net.Address.parseIp("127.0.0.1", 7000);

    var sendFrame = async send_message(addr);
    // ... do something else while
    //     the message is being sent ...
    try await sendFrame;
}

// Note how the function definition doesn't require any static
// `async` marking. The compiler can deduce when a function is
// async based on its usage of `await`.
fn send_message(addr: net.Address) !void {
    // We could also delay `await`ing for the connection
    // to be established, if we had something else we
    // wanted to do in the meantime.
    var socket = net.tcpConnectToAddress(addr);
    defer socket.close();

    // Using both await and async in the same statement
    // is unnecessary and non-idiomatic, but it shows
    // what's happening behind the scenes when `io_mode`
    // is `.evented`.
    _ = try await async socket.write("Hello World!\n");
}
```

By using the `async` keyword, you're creating the coroutine and running it until it encounters a suspension point (when it has to wait for I/O to happen, roughly

speaking). The return value is what Zig calls an "async frame" and is to some degree equivalent to `Future`, `Task`, `Promise`, or `Coroutine` objects from other languages.

One final trick

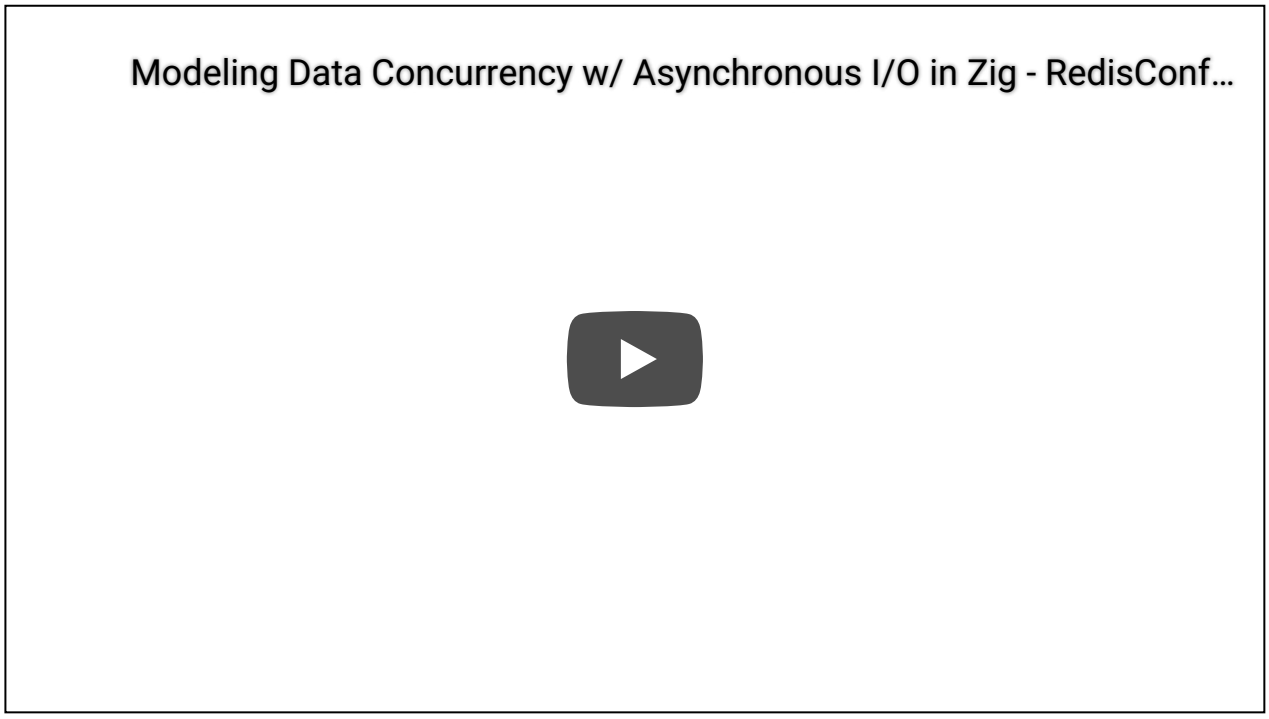
Let me show you the final trick that completes the puzzle: using async/await in blocking mode. To revert back to blocking I/O, all we have to do is delete the special declaration we added in the beginning (or use the corresponding enum case).

```zig
const net = @import("std").net;

pub const io_mode = .blocking;

pub fn main() !void {
    const addr = try net.Address.parseIp("127.0.0.1", 7000);

    // yes, this still works
    var sendFrame = async send_message(addr);
    try await sendFrame;
}

fn send_message(addr: net.Address) !void {
    var socket = net.tcpConnectToAddress(addr);
    defer socket.close();

    // this too
    _ = try await async socket.write("Hello World!\n");
}
```

Yes, this program compiles and works as expected. The function is no longer async, and in fact both keywords basically become no-ops, but **the point is that you can express concurrency even if you're not able to take advantage of it**. This might not seem such a big deal in our small example, but it's the principle that allows libraries to offer both blocking and evented I/O capabilities off of a single code base.

A concrete example

A while ago I started working on OkRedis, a Redis client library written in Zig that tries to offer as many niceties it can to the user without compromising in terms of efficiency. Among other things, it features full support for both blocking and evented I/O in a single codebase. If you want to learn more, take a look at the available documentation on GitHub and watch this talk Andrew Kelley (the creator of Zig) co-authored with me. In it, Andrew explains the basics of async/await in Zig and, in the second part, I demo OkRedis.



"Modeling Data Concurrency with Asynchronous I/O in Zig"

Understanding the limits

While the compiler is very smart and defeating function coloring has many practical benefits, it's not a magic bullet, so let me immediately demystify some ideas about it.

Q: WILL ENABLING EVENTED I/O IMMEDIATELY MAKE MY PROGRAM FASTER?

No, to make your program take advantage of evented I/O you need to express concurrency in your code. If you never did that work, enabling evented I/O will not provide any appreciable advantage, but you might experience better

performance if one of the libraries you're using has been properly designed for async/await.

That said if your code gets embedded in a larger project that does make use of async/await, the automated translation to evented I/O will make your code play nice with the surrounding context, to some extent.

Q: SO ALL ASYNC APPLICATIONS CAN BE MADE BLOCKING JUST BY FLIPPING A SWITCH?

No, there's plenty of applications that need evented I/O to behave correctly. Switching to blocking mode, without introducing any change, might cause them to deadlock. Think for example of a single application that acts both as server and client to itself.

That said, during compile-time, it's possible to inspect if the overall program is in evented mode or not, and properly designed code might decide to move to a threaded model when in blocking mode, for example.

Q: SO I DON'T EVEN HAVE TO THINK ABOUT NORMAL FUNCTIONS VS COROUTINES IN MY LIBRARY?

No, occasionally you will have to. As an example, if you're allowing your users to pass to your library function pointers at runtime, you will need to make sure to use the right calling convention based on whether the function is async or not. You normally don't have to think about it because the compiler is able to do the work for you at compile-time, but that can't happen for runtime-known values.

The silver lining is that you have at your disposal all the tools to account for all the possibilities in a simple and clear way. Once you get the details right, the code will be no more complicated than it has to be, and your library will be easy to use.

Concurrency and resource allocation

While I wrote the introduction with the average developer in mind, you need to be aware that Zig is not a dynamically typed language and, on top of that, it puts in your hands a lot of power (and responsibility) when it comes to resource management. If you know how to do async/await in C#, JavaScript, or Python, for example, you won't be able to immediately know how to do *everything* in Zig.

In particular, garbage-collected languages hide from you where the memory is coming from. This makes things much easier for the programmer, but the price for this extra ease is shouldered entirely by the machine. This is nothing new and it's a trade-off very often worth doing but, especially when it comes to async/await, it's problematic because you lose control of how much memory you're consuming, eventually over-committing and encountering issues when under heavy load (see this blog post for more info).

One of the main points of Zig is to make resource allocation always clear and manageable. When it comes to async/await, this means that all the memory required to run a coroutine is represented by its underlying async frame. Once you have a frame (be it because it's static memory or because the corresponding dynamic allocation succeded), then you know the coroutine will be able to run to completion without problems. In the context of a HTTP server, for example, this means that you will be able to know upfront if you have enough resources to accept a connection or not, without encountering unrecoverable error conditions.

A final word on concurrency in Zig

Up until now, I've only talked about the language features that implement coroutines. I haven't mentioned all of them but, more importantly, I haven't talked about the event loop. In Zig, the event loop is part of the standard library and the idea is to make it swappable.

At the moment of writing, there's still a lot of work to do on the event loop, but you can already try everything out today. The current implementation is already multi-threaded, in case you were wondering. Just go to ziglang.org, download the latest version, and take a look a the docs.

...AND WHAT ABOUT GO?

Ah, yes. Go — alongside a few other languages in fact — doesn't have function coloring problems. Andrew mentions Go in his part of the aforelinked talk, but I'll give my two cents here.

If you've read my blog before, you know that I like Go. I think that goroutines are generally preferable to async/await in application-level programming, especially when it comes to server-side applications, which I believe to be Go's main domain.

I think goroutines are preferable in that context because async/await is a much lower level tool that is easy to misuse, but when it comes to writing code with critical requirements of correctness and efficiency, you need async/await and you need Zig's philosophy that we all should strive to write **robust**, **optimal**, and **reusable** software.

Thank you very much for reading! Got a comment? Tweet @croloris.

P.S. I started live coding regularly on Twitch, follow the channel to be notified when I go live next!

← The Upcoming Future of Online Meetups Addio Redis, I'm leaving Redis Labs →