

Rust's Journey to Async/Await

Steve Klabnik



Hi, I'm Steve!

- On the Rust team
- Work at Cloudflare
- Doing two workshops!



bateaux

without

gâteaux

on

cakes

please



Following

without butts, dreams dry up

@withoutboats Follows you

love and rage

 Joined March 2015

267 Following **2,023** Followers



Followed by Rust Secure Code WG, David Tolnay, and 194 others you follow

What is async?

Parallel: do multiple things at once

Concurrent: do multiple things, not at once

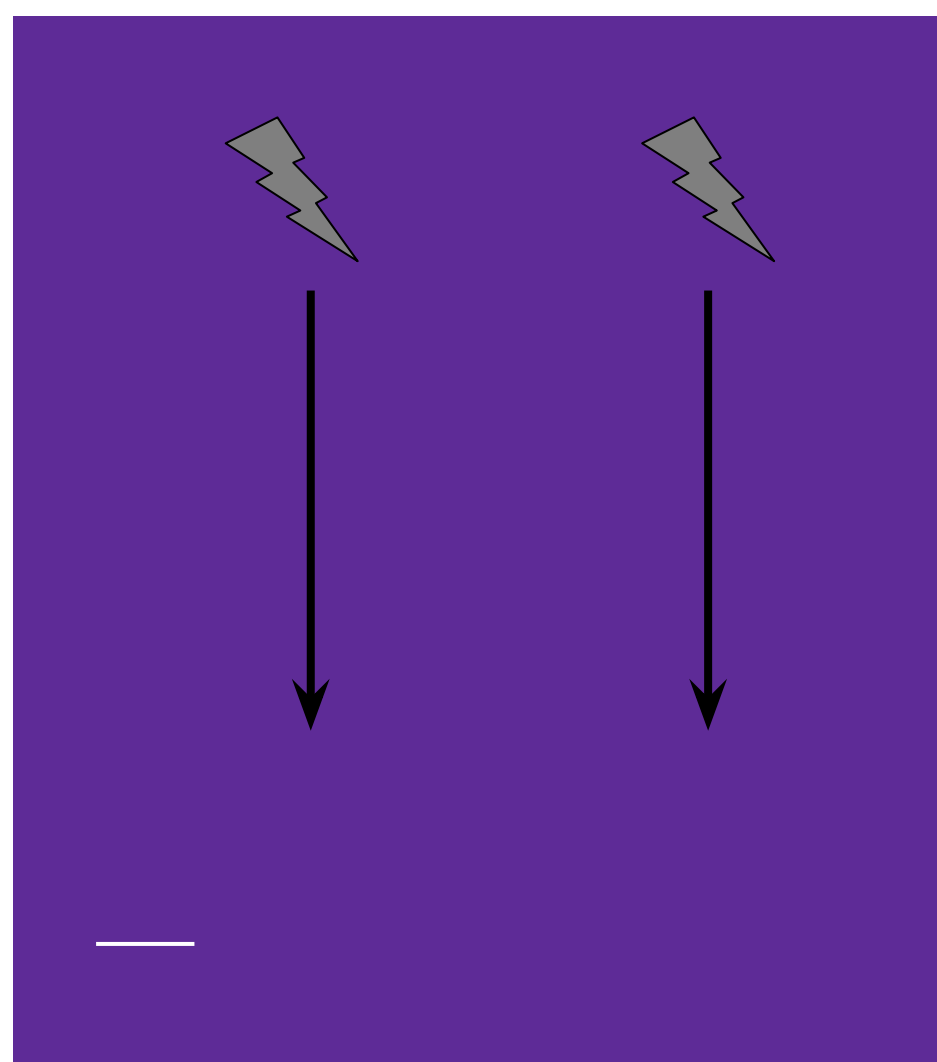
Asynchronous: actually unrelated! Sort of...

“Task”

A generic term for some computation running in a parallel or concurrent system

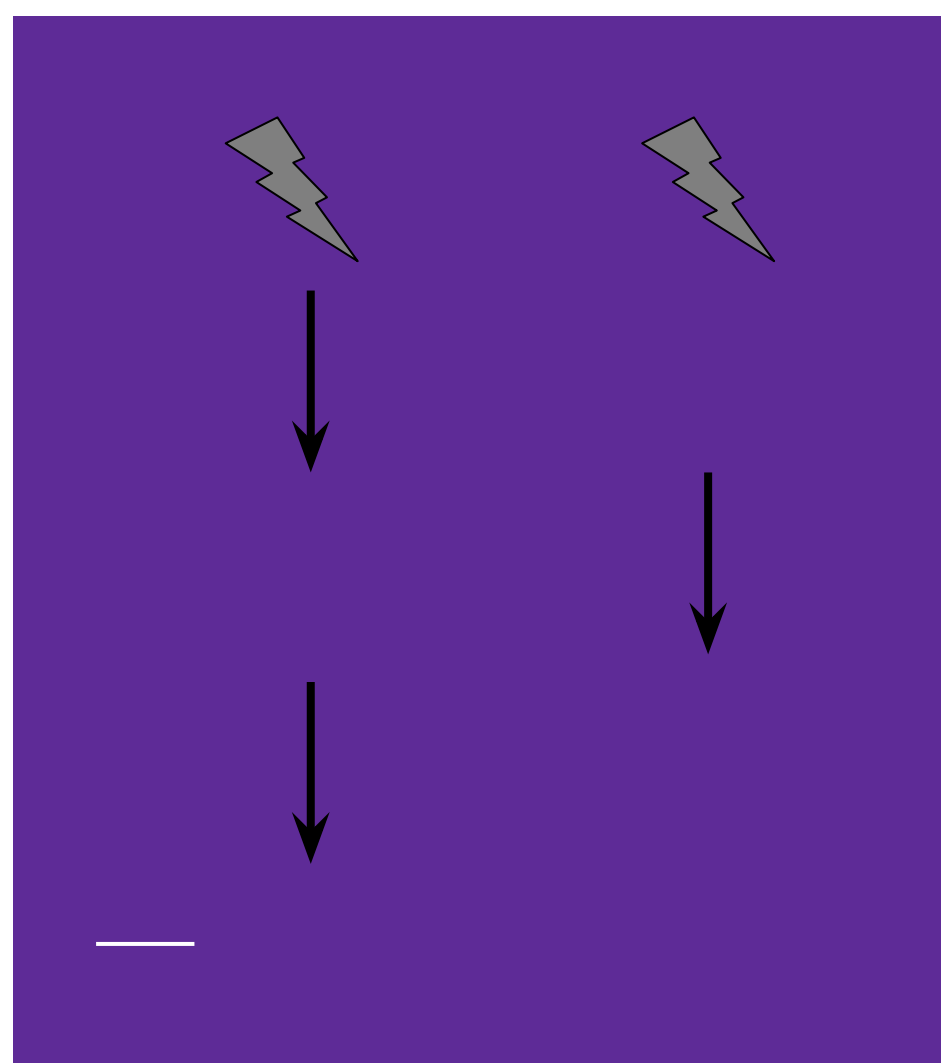
Parallel

**Only possible with multiple
cores or CPUs**



Concurrent

Pretend that you have multiple
cores or CPUs



Asynchronous

**A word we use to describe
language features that enable
parallelism and/or
concurrency**

**Even more
terminology**

Cooperative vs Preemptive Multitasking

Cooperative Multitasking

**Each task decides when to
yield to other tasks**

Preemptive Multitasking

**The system decides when to
yield to other tasks**

Native vs green threads

Native threads

Sometimes called “1:1 threading”

**Tasks provided by the
operating system**

Green Threads

Sometimes called “N:M threading”

**Tasks provided by your
programming language**

Native vs Green threads

Native thread advantages:

- Part of your system; OS handles scheduling
- Very straightforward, well-understood

Native thread disadvantages:

- Defaults can be sort of heavy
- Relatively limited number you can create

Green thread advantages:

- Not part of the overall system; runtime handles scheduling
- Lighter weight, can create many, many, many, many green threads

Green thread disadvantages:

- Stack growth can cause issues
- Overhead when calling into C

Why do we care?

The C10K problem

[\[Help save the best Linux news source on the web -- subscribe to Linux Weekly News!\]](#)

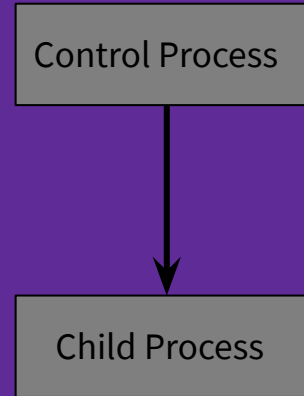
It's time for web servers to handle ten thousand clients simultaneously, don't you think? After all, the web is a big place now.

And computers are big, too. You can buy a 1000MHz machine with 2 gigabytes of RAM and an 1000Mbit/sec Ethernet card for \$1200 or so. Let's see - at 20000 clients, that's 50KHz, 100Kbytes, and 50Kbits/sec per client. It shouldn't take any more horsepower than that to take four kilobytes from the disk and send them to the network once a second for each of twenty thousand clients. (That works out to \$0.08 per client, by the way. Those \$100/client licensing fees some operating systems charge are starting to look a little heavy!) So hardware is no longer the bottleneck.

In 1999 one of the busiest ftp sites, cdrom.com, actually handled 10000 clients simultaneously through a Gigabit Ethernet pipe. As of 2001, that same speed is now [being offered by several ISPs](#), who expect it to become increasingly popular with large business customers.

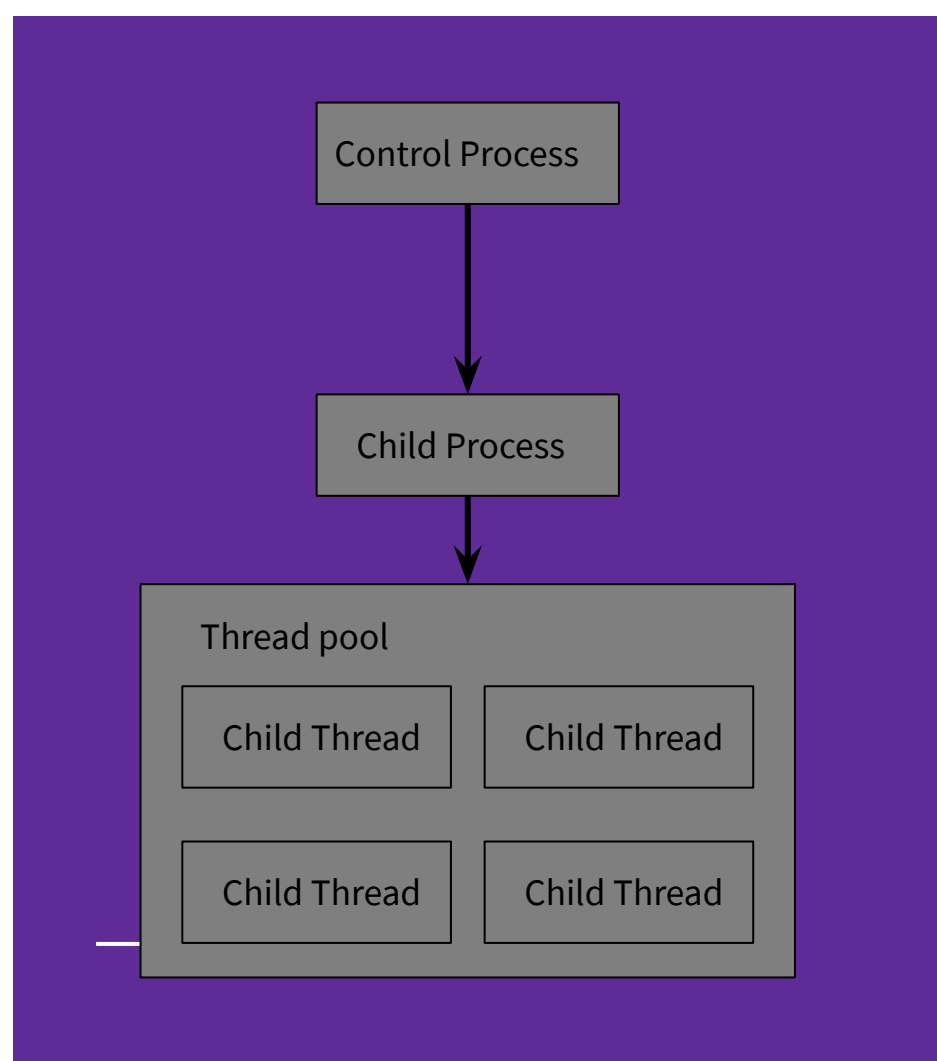
Apache

“Pre-fork”



Apache

“worker”



**Let's talk about
Rust**

**Rust was built to
enhance Firefox,
which is an HTTP
client, not server**



Search documentation...

Module **green**

The "green scheduling" library

This library provides M:N threading for rust programs. Internally this has the implementation of a green scheduler and allocation strategy.

This can be optionally linked in to rust programs in order to provide M:N functionality inside of 1:1 programs.

MODULES

basic This is a basic event loop implementation not meant for any "real purposes" other than testing the

context



std::io

MODULES

- buffered
- comm_adapters
- extensions
- flate
- fs
- io_error
- mem
- net**
- pipe

Search documentation...

Module `std::io::net`

Synchronous, non-blocking network I/O. Synchronous, non-blocking network I/O.

REEXPORTS

```
pub use self::addrinfo::get_host_addresses;
```

MODULES

- addrinfo** Synchronous DNS Resolution
- ip**
- tcp**
- udp**
- unix** Named pipes

**“Synchronous,
non-blocking
network I/O”**

**Isn't this a
contradiction in
terms?**

	Synchronous	Asynchronous
Blocking	Old-school implementations	Doesn't make sense
Non-blocking	Go, Ruby	Node.js

Tons of options

Synchronous, blocking

- Your code looks like it blocks, and it does block
- Very basic and straightforward

Asynchronous, non-blocking

- Your code looks like it doesn't block, and it doesn't block
- Harder to write

Synchronous, non-blocking

- Your code looks like it blocks, but it doesn't!
- The secret: the runtime is non-blocking
- Your code still looks straightforward, but you get performance benefits
- A common path for languages built on synchronous, blocking I/O to gain performance while retaining compatibility

**Not all was well in
Rust-land**



Docs (Nightly)

[Book](#)

[Reference](#)

[API docs](#)

[All docs](#)

Docs (Alpha)

[Book](#)

[Reference](#)

[API docs](#)

[All docs](#)

Rust is a systems programming language that runs blazingly fast, prevents almost all crashes*, and eliminates data races.

[Show me more!](#)

A “systems programming language” that doesn’t let you use the system’s threads?



Crates

|Click or press 'S' to search, '?' for more options...

Crate **native** | experimental

The native I/O and threading crate

This crate contains an implementation of 1:1 scheduling for a "native" runtime. blocking version of I/O.

Starting with libnative

oc

ena

llections



Crates

|Click or press 'S' to search, '?' for more options...

Crate **green** | **experimental**

The "green scheduling" library

This library provides M:N threading for rust programs. Inter-process switching and a stack-allocation strategy. This can be optimized for high-performance programs.

Architecture

oc

ena

llections

In today's Rust, there is a single I/O API -- `std::io` -- that provides blocking operations only and works with both threading models. Rust is somewhat unusual in allowing programs to mix native and green threading, and furthermore allowing *some* degree of interoperability between the two. This feat is achieved through the runtime system -- `librustrt` - which exposes:

- The `Runtime` trait, which abstracts over the scheduler (via methods like `deschedule` and `spawn_sibling`) as well as the entire I/O API (via `local_io`).
- The `rtio` module, which provides a number of traits that define the standard I/O abstraction.
- The `Task` struct, which includes a `Runtime` trait object as the dynamic entry point into the runtime.

In this setup, `libstd` works directly against the runtime interface. When invoking an I/O or scheduling operation, it first finds the current `Task`, and then extracts the `Runtime` trait object to actually perform the operation.

**Not all was well in
Rust-land**

Summary

This RFC proposes to remove the *runtime system* that is currently part of the standard library, which currently allows the standard library to support both native and green threading. In particular:

- The `libgreen` crate and associated support will be moved out of tree, into a separate Cargo package.
- The `librustrt` (the runtime) crate will be removed entirely.
- The `std::io` implementation will be directly welded to native threads and system calls.
- The `std::io` module will remain completely cross-platform, though *separate* platform-specific modules may be added at a later time.

**Rust 1.0 was
approaching**

**Ship the minimal
thing that we
know is good**

**Rust 1.0 was
released!** 🎉

**... but still, not all
was well in
Rust-land**

People  Rust

**People want to
build network
services in Rust**

**Rust is supposed
to be a
high-performance
language**

**Rust's I/O model
feels retro, and
not performant**

**The big problem
with native
threads for I/O**

CPU bound vs I/O bound

CPU Bound

My processor is working hard

**The speed of completing a task
is based on the CPU crunching
some numbers**

I/O Bound

Doing a lot of networking

The speed of completing a task is based on doing a lot of input and output

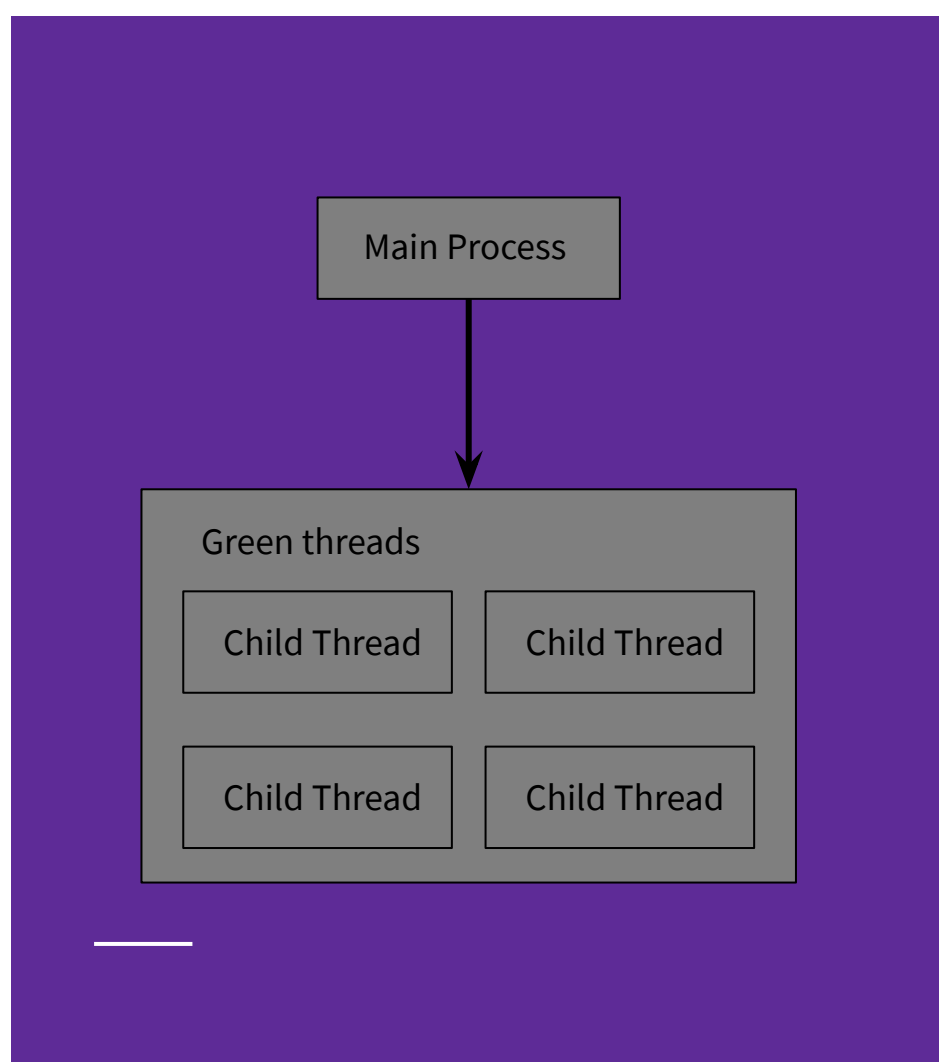
**When you're
doing a lot of I/O,
you're doing a lot
of waiting**

**When you're
doing a lot of
waiting, you're
tying up system
resources**

Go

Asynchronous I/O
with green threads

(Erlang does this too)




Native vs Green threads

PREVIOUSLY

Native thread advantages:

- Part of your system; OS handles scheduling
- Very straightforward, well-understood

Native thread disadvantages:

- Defaults can be sort of heavy
- Relatively limited  create

Green thread advantages:

- Not part of the overall system; runtime handles scheduling
- Lighter weight, can create many, many, many, many green threads

Green thread disadvantages:

- Stack growth can cause issues
- Overhead when calling into C

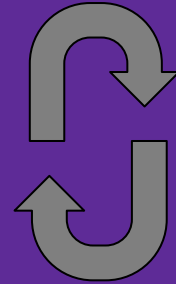
A “systems programming language” that has overhead when calling into C code?

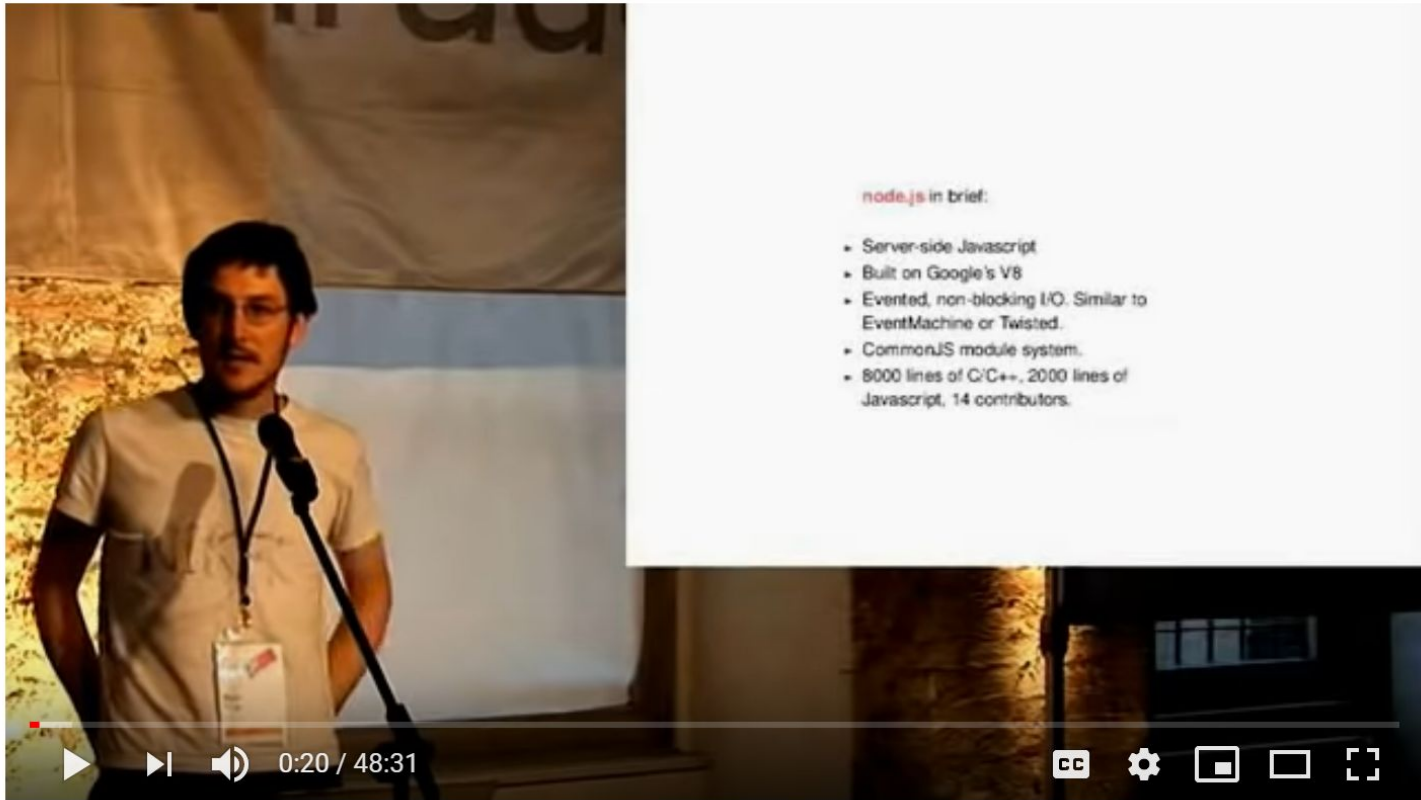
**Luckily, there is
another way**

Nginx

Asynchronous I/O

Event Loop





node.js in brief:

- Server-side Javascript
- Built on Google's V8
- Evented, non-blocking I/O. Similar to EventMachine or Twisted.
- CommonJS module system.
- 8000 lines of C/C++, 2000 lines of Javascript, 14 contributors.

Ryan Dahl: Original Node.js presentation

166,252 views



2.5K



27



SHARE



SAVE



**Evented I/O
requires
non-blocking APIs**

Blocking vs non-blocking

Using the File System module as an example, this is a **synchronous** file read:

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
```

And here is an equivalent **asynchronous** example:

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
});
```

“Callback hell”

Callback Hell

A guide to writing asynchronous JavaScript programs

What is "callback hell"?

Asynchronous JavaScript, or JavaScript that uses callbacks, is hard to get right intuitively. A lot of code ends up looking like this:

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }).bind(this)
        }
      })
    })
  }
})
```

A **Promise** is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a *promise* to supply the value at some point in the future.

A **Promise** is in one of these states:

- *pending*: initial state, neither fulfilled nor rejected.
- *fulfilled*: meaning that the operation completed successfully.
- *rejected*: meaning that the operation failed.

Promises

```
let myFirstPromise = new Promise((resolve, reject) => {  
  setTimeout(function(){  
    resolve("Success!");  
  }, 250);  
});
```

```
myFirstPromise.then((successMessage) => {  
  console.log("Yay! " + successMessage);  
});
```

Promises

```
let myFirstPromise = new Promise((resolve, reject) => {  
  setTimeout(function(){  
    resolve("Success!");  
  }, 250);  
});
```

```
myFirstPromise.then((successMessage) => {  
  console.log("Yay! " + successMessage);  
}).then((...) => {  
  //  
}).then((...) => {  
  //  
});
```

Your Server as a Function

Marius Eriksen

Twitter Inc.

marius@twitter.com

Abstract

Building server software in a large-scale setting, where systems exhibit a high degree of concurrency and environmental variability, is a challenging task to even the most experienced programmer. Efficiency, safety, and robustness are paramount—goals which have traditionally conflicted with modularity, reusability, and flexibility.

We describe three abstractions which combine to present a powerful programming model for building safe, modular, and efficient server software: Composable *futures* are used to relate concurrent, asynchronous actions; *services* and *filters* are specialized functions

Services Systems boundaries are represented by asynchronous functions called *services*. They provide a symmetric and uniform API: the same abstraction represents both clients and servers.

Filters Application-agnostic concerns (e.g. timeouts, retries, authentication) are encapsulated by *filters* which compose to build services from multiple independent modules.

Server operations (e.g. acting on an incoming RPC or a timeout) are defined in a declarative fashion, relating the results of the (possibly many) subsequent sub-operations through the use of fu

Aaron Turon

[Archive](#)

[Feed](#)

Zero-cost futures in Rust

11 Aug 2016

One of the key gaps in Rust's ecosystem has been a strong story for fast and productive *asynchronous I/O*. We have solid foundations, like the [mio](#) library, but they're very low level: you have to wire up state machines and juggle callbacks directly.

We've wanted something higher level, with better ergonomics, but also better *composability*, supporting an ecosystem of asynchronous abstractions that all work together. This story might sound familiar: it's the same goal that's led to the introduction of *futures* (aka promises) in [many languages](#), with some supporting *async/await* sugar on top.

A major tenet of Rust is the ability to build [zero-cost abstractions](#), and that leads to one additional goal for our async I/O story: ideally, an abstraction like futures should compile down to something equivalent to the state-machine-and-callback-juggling code we're writing today (with no additional runtime overhead).

Futures 0.1

```
pub trait Future {
    type Item;
    type Error;

    fn poll(&mut self) -> Poll<Self::Item, Self::Error>;
}

id_rpc(&my_server).and_then(|id| {
    get_row(id)
}).map(|row| {
    json::encode(row)
}).and_then(|encoded| {
    write_string(my_socket, encoded)
})
```

Promises and Futures are different!

- Promises are built into JavaScript
 - The language has a runtime
 - This means that Promises start executing upon creation
 - This feels simpler, but has some drawbacks, namely, lots of allocations
- Futures are not built into Rust
 - The language has no runtime
 - This means that you must submit your futures to an executor to start execution
 - Futures are inert until their `poll` method is called by the executor
 - This is slightly more complex, but **extremely** efficient; a single, perfectly sized allocation per task!
 - Compiles into the state machine you'd write by hand with evented I/O

Futures 0.1: Executors

```
use tokio;

fn main() {
    let addr = "127.0.0.1:6142".parse().unwrap();
    let listener = TcpListener::bind(&addr).unwrap();

    let server = listener.incoming().for_each(|socket| {
        Ok(())
    })
    .map_err(|err| {
        println!("accept error = {:?}", err);
    });

    println!("server running on localhost:6142");

    tokio::run(server);
}
```

**We used
Futures 0.1 to
build stuff!**

**The design had
some problems**

Futures 0.2

```
trait Future {  
    type Item;  
    type Error;  
  
    fn poll(&mut self, cx: task::Context) ->  
Poll<Self::Item, Self::Error>;  
}
```

No implicit context, no more need for thread local storage.

↑ **aturon** rust 36 points · 1 year ago



Would you suggest that the ecosystem goes through two breaking changes (now and for 0.3) or should libraries like Tokio maintain support for both 0.1 and 0.2 and then have a single breaking change when 0.3 is released.

The latter. I consider 0.2 a "snapshot" that's good for experimentation but shouldn't be used heavily, since stable 0.3 should be coming in a couple of months or less.

Give Award **Share** **Report** **Save**

↑ **sdroege_** 28 points · 1 year ago



This seems to be counterproductive. If you want people to experiment with the changes they need their dependencies using futures 0.2. Otherwise any experimentation would only be possible for completely standalone things and not even on top of tokio or hyper, and that would limit the amount of feedback you get a lot. Especially with regards to usability.



aturon

rust



36 points · 1 year ago



Would you suggest that the ecosystem goes through two breaking changes (now and for 0.3) or should libraries like Tokio maintain support for both 0.1 and 0.2 and then have a single breaking change when 0.3 is released.

The latter. I consider 0.2 a "snapshot" that's good for experimentation but shouldn't be used heavily, since stable 0.3 should be coming in a couple of months or less.

[Give Award](#) [Share](#) [Report](#) [Save](#)



sdroege_ 28 points · 1 year ago



This seems to be counterproductive. If you want people to experiment with the changes they need their dependencies using futures 0.2. Otherwise any experimentation would only be possible for completely standalone things and not even on top of tokio or hyper, and that would limit the amount of feedback you get a lot. Especially with regards to usability.

Async/await

```
// with callback
request('https://google.com/', (response) => {
  // handle response
})

// with promise
request('https://google.com/').then((response) => {
  // handle response
});

// with async/await
async function handler() {
  let response = await request('https://google.com/')
  // handle response
}
```

**Async/await lets you
write code that feels
synchronous, but is
actually asynchronous**

Async/await is more important in Rust than in other languages because Rust has no garbage collector

Rust example: synchronous

```
fn read(&mut self, buf: &mut [u8]) -> Result<usize, io::Error>

let mut buf = [0; 1024];
let mut cursor = 0;

while cursor < 1024 {
    cursor += socket.read(&mut buf[cursor..])?;
}
```

Rust example: async with Futures

```
fn read<T: AsMut<[u8]>>(self, buf: T) ->  
    impl Future<Item = (Self, T, usize), Error = (Self, T, io::Error)>
```

... the code is too big to fit on the slide

The main problem: the borrow checker doesn't understand asynchronous code.



The constraints on the code when it's created and when it executes are different.

Rust example: async with async/await

```
async {  
    let mut buf = [0; 1024];  
    let mut cursor = 0;  
  
    while cursor < 1024 {  
        cursor += socket.read(&mut buf[cursor..]).await?;  
    };  
  
    buf  
}
```

`async/await` can teach the borrow checker about these constraints.

Not all futures can error

```
trait Future {  
    type Item;  
    type Error;   
  
    fn poll(&mut self, cx: task::Context) ->  
    Poll<Self::Item, Self::Error>;   
}
```


std::future

```
pub trait Future {  
    type Output;  
  
    fn poll(self: Pin<&mut Self>, cx: &mut Context)  
-> Poll<Self::Output>;  
}
```

Pin is how `async/await` teaches the borrow checker.

If you need a future that errors, set `Output` to a `Result<T, E>`.

**... but one more
thing...**

What syntax for async/await?

async is not an issue

JavaScript and C# do:

```
await value;
```

But what about ? for error handling?

```
await value?;
```

```
await (value?);
```















```
(await value)?;
```

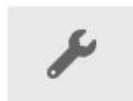
What syntax for async/await?

What about chains of await?

```
(await (await value)?);
```

We argued and
argued and
argued and
argued and
argued and ar...

- 

 Niko Matsakis via R. 11
 Rust
 [rust-internals] [language design] Async-await experience reports - nikon
- 

 Isaac Whitfield via. 25
 Rust
 [rust-internals] [language design] Does `await` truly need fixing? - Centril
- 

 Daboross via Rust I. 35
 Rust
 [rust-internals] [language design] Pre-RFC: Add language support for glo
- 

 Nemo157 via Rust In.
 Rust
 [rust-internals] [language design] A final proposal for await syntax - Nem
- 

 Brian West via Rust. 100
 Rust
 [rust-internals] [language design] A final proposal for await syntax - tkaite
- 

 Elahn lentile via R. 67
 Rust
 [rust-internals] [language design] Async / Await syntax straw poll - stever
- 

 Soni via Rust Inter. 8
 Rust
 [rust-internals] Replace



May 6

1 / 478

May 6



Apr 29

1 / 212

Apr 29

async/await: dot-aw

r/rust · Posted by u/mique

  **168 Comments**

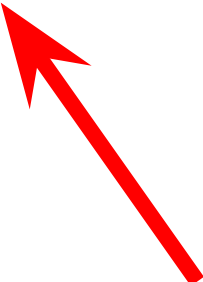
async/await Syntax

r/rust · Posted by u/cramer

  **133 Comments**

What syntax for async/await?

```
async {  
  let mut buf = [0; 1024];  
  let mut cursor = 0;  
  
  while cursor < 1024 {  
    cursor += socket.read(&mut buf[cursor..]).await?;  
  };  
  
  buf  
}  
  
// no errors  
future.await  
// with errors  
future.await?
```



... there's actually
even one last
issue that's
popped up

**... this talk is
already long
enough**

Additional Ergonomic improvements

```
use runtime::net::UdpSocket;

#[runtime::main]
async fn main() -> std::io::Result<()> {
    let mut socket = UdpSocket::bind("127.0.0.1:8080"?);
    let mut buf = vec![0u8; 1024];

    println!("Listening on {}", socket.local_addr()?);

    loop {
        let (recv, peer) = socket.recv_from(&mut buf).await?;
        let sent = socket.send_to(&buf[..recv], &peer).await?;
        println!("Sent {} out of {} bytes to {}", sent, recv, peer);
    }
}
```

WebAssembly?

```
#[wasm_bindgen]
pub fn wasm_entry(path: String, data: Data) -> Promise {
    future_to_promise(async move {
        let path = PathBuf::from(path);

        let future = JsFuture::from(data.get(path.to_str().unwrap()));
        let contents = future
            .await
            .expect("couldn't fetch page data")
            .as_string()
            .expect("couldnt get a string");

        let response = Response {
            body,
            status_code: 200,
            content_type,
        };

        JsValue::from_serde(&response).map_err(|_| JsValue::from_str("cou
    })
}
```

WebAssembly?

```
#[wasm_bindgen]
pub fn wasm_entry(path: String, data: Data) -> Promise {
    future_to_promise(async move {
        let path = PathBuf::from(path);

        let future = JsFuture::from(data.get(path.to_str().unwrap()));
        let contents = future
            .await
            .expect("couldn't fetch page data")
            .as_string()
            .expect("couldnt get a string");

        let response = Response {
            body,
            status_code: 200,
            content_type,
        };

        JsValue::from_serde(&response).map_err(|_| JsValue::from_serde("couldnt get a string"))
    })
}
```

Promise

Future

Promise

**Finally landing in
Rust 1.37**

Or maybe 1.38

**Finally landing in
Rust 1.37**

Or maybe 1.38



without butts, dreams dry up

@withoutboats



Storm warning: comment hurricane incoming on the rust repo.
We are stabilizing async/await



[Stabilization] async/await MVP · Issue #62149 · rust-lang/rust
Stabilization target: 1.38.0 (beta cut 2019-08-15) Executive Summary
This is a proposal to stabilize a minimum viable async/await feature, ...
[github.com](#)

7:47 AM · Jun 26, 2019 · [Twitter for iPhone](#)

29 Retweets **95** Likes

**Finally landing in
Rust 1.38!!!!1**

JSON serialization

Single query

Multiple queries

Fortunes

Data updates

Plaintext

Fortunes

Best (bar chart)

Data table

Latency

Framework overhead

Best fortunes responses per second, Test environment (368 tests)

[Rnk](#) Framework

Best performance (higher is better)

1	 actix-core	699,975		100.0%
2	 actix-pg	630,441		90.1%
3	 atreugo-prefork-quicktemplate	435,042		62.2%

**Lesson: a
world-class
I/O system
implementation
takes years**

**Lesson: different
languages have
different
constraints**

Thank you!

@steveklabnik