node JS Interactive

October 4-6, 2017 | Vancouver, BC

High Performance Javascript With Rust

Amir Yasin, Senior Engineer, 2U

About me

Amir Yasin

I’ve been passionate about writing software for over 20 years. I’ve worked in a number of industries doing everything from software on embedded controllers to websites. I currently work for an amazing education company called 2U.

Twitter: @ayasin   Medium: @ayasin

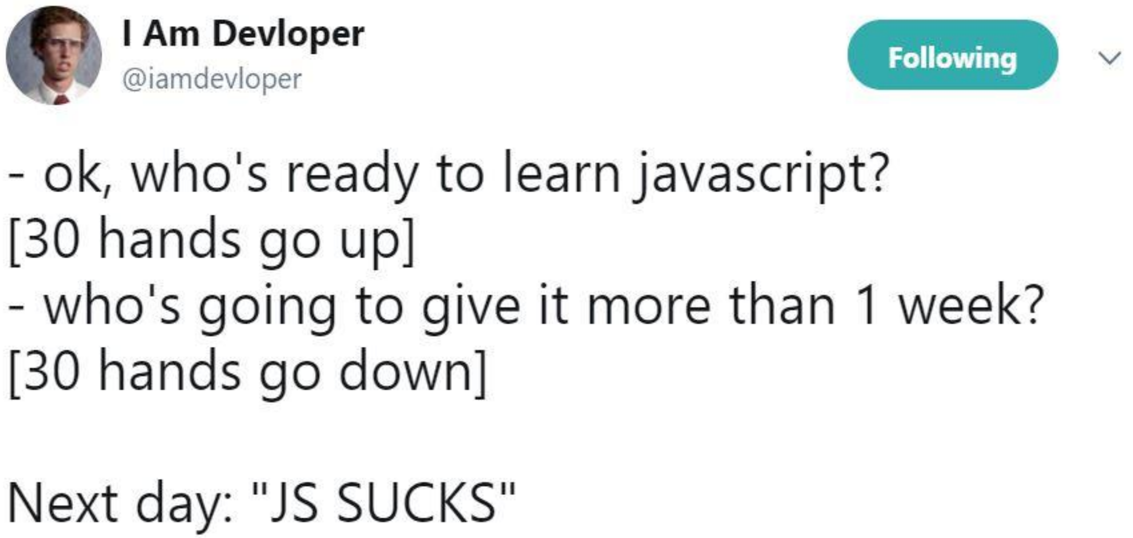
What I hope you get out of this talk

- A basic understanding of Rust (enough to take advantage of the performance boost)
- An understanding of how to call Rust from Node
- An understanding of when it's worth it to use Rust to boost your performance

October 4-6, 2017 | Vancouver, BC

What you won’t get out of this talk

You aren’t going to become a Rust expert in 30 min


I Am Devloper
@iamdevloper
Following

- ok, who's ready to learn javascript?
[30 hands go up]
- who's going to give it more than 1 week?
[30 hands go down]

Next day: "JS SUCKS"

Let’s get started…What is Rust?

- Rust is a low level (systems) programming language with high level language convenience
- Strongly typed
- Sponsored by Mozilla, used by many including NPM, Coursera, and Dropbox
- Designed for speed and safety
- Rust’s Motto is Hack Without Fear

Speed (AKA why you're here)

- Rust is compiled
- Rust has no garbage collection or automated resource management so it's speed is predictable
- Rust lets you get very close to the hardware
- Rust is as fast as C in many cases (in cases where it's not, it's a very close second)

October 4-6, 2017 | Vancouver, BC

Safety (a really unique feature)

- No null pointer or use after free problems
- No null at all!
- No manual memory management via a very interesting memory “ownership” model that’s checked at compile time
- Threading made easier through the compile time elimination of data races
- You can even use the compiler to prevent code that leaks “secrets” from compiling

Mine, yours, and ours

What is ownership?

The right to destroy

node JS® Interactive

October 4-6, 2017 | Vancouver, BC

Ownership

Memory safety continued

- Variables are immutable by default
- Functions or other bits of code can “borrow” variables you own
- You can mark variables as mutable
- You can have many “borrows” out, but only one “mutable borrow”
- You can also transfer ownership to someone else, but then you can’t use the variable anymore unless you borrow it from them

October 4-6, 2017 | Vancouver, BC

More about memory

- Because of this unique model, the compiler can ensure that memory can’t be corrupted as long as you don’t use “unsafe” (an escape hatch that allows you to interact with some C libraries)
- This means you get as much or more safety than higher languages without any of the runtime overhead.

Enough about memory, teach me Rust!

The easiest way to do this is to map it to something we already know (like Javascript!)

Functions

Javascript

```
function sayHello() { console.log('Hello world!'); }
```

Rust

```
fn say_hello() -> () { print!("Hello world!"); }
```

In Rust, every function returns a value. In this case it's ()

Variables

Javascript

```
var x;
let x1 = 3;
const y = 2;
```

Rust

```
let x = 12;
let mut x1 = 22;
let y: f64 = 34;
```

This is how we tell Rust a variable can change

This is a type specification

Flow control

Javascript

```
if (x) { /* do this */ } else { /* do that */ }
let g = x ? 1 : 2;
```

Rust

```
if (x) { /* do this */ } else { /* do that */ }
let g = if (x) { 1 } else { 2 };
```

Notice there's no ;

Flow control

October 4-6, 2017 | Vancouver, BC

Javascript

```
switch (x) {
    case 1: /* do this */ break;
   default: /* do that */
}
```

Rust

```
match x {
  1 => /* do this */,
  _ => /* do that */,
}
```

This is the equivalent of break

Matches anything (default)

Looping

October 4-6, 2017 | Vancouver, BC

Javascript

```
while (x) { /* do some stuff */ }
do { /* stuff */ } while (x)
for (var i = 0; i < 10; i++) { /*do something 10 times*/ }
```

Rust

```
while (x) { /* do some stuff */ }
for i in 0..10 { /* do something 10 times */ }
for thing in coll { /* do some thing (get it?) */ }
loop { /* do this forever */}
```

Types

Javascript

```
// yeah, about that...
```

Rust

```
bool i8 i16 i32 i64 u8 ...
string array slice tuple
Vector HashMap
```

Not core, but part of std lib

Packaging

- Javascript
  - npm/yarn
  - package.json/yarn.lock
  - import/require
- Rust
  - cargo
  - cargo.toml/cargo.lock
  - use

Let’s solve a computationally intensive problem

Given 2 large numbers, find the least common factor (lcf). If there is no common factor, return 0.

Installing the Rust toolchain

Probably the easiest toolchain you've ever installed

www.rustup.rs

Just click through the steps

Getting ready to code some Rust

```
cargo new <app-or-lib-name>
```

By default this creates a lib. Sometimes it's useful to make an app to test to make it easy to debug the functions. To do this add the `--app` flag

Modify cargo.toml

We'll need to add a couple of lines to our `cargo.toml`:

```
[lib]
crate-type = ["dylib"]
```

We want our lib loaded at runtime

Before we write some Rust

Let’s prepare the JS side, and write a JS version

Install JS dependencies

```
npm install ffi
```

JS version

```
function lcf(num1, num2) {
  const stop = num1 > num2 ? num2 + 1 : num1 + 1;
  for (let check = 2; check < stop; check++) {
    if (!(num1 % check) && !(num2 % check)) {
      return check;
    }
  }
  return 0;
}
```

OMG it's happening! Rust code!

October 4-6, 2017 | Vancouver, BC

```
#[no_mangle]
pub extern fn lcf(n1: i64, n2: i64) -> i64 {
  let stop = if n1 < n2 { n1 + 1 } else { n2 + 1 };
  for check in 2..stop {
    if (n1 % check == 0) && (n2 % check == 0) {
      return check;
    }
  }
  0
}
```

Remember Rust is compiled; cargo to the rescue!

In the folder cargo created earlier, run

```
cargo build --release
```

Once complete, you should have a `target/release` folder containing a lib.

Connecting Rust and Node

```
const ref = require('ref’);
const ffi = require('ffi’);

const int64 = ref.types.int64;

const rust = ffi.Library(path.join(__dirname, 'rust-lcf/target/release/liblcf'), {
    lcf: [int64, [ int64, int64 ]],
});

console.log(rust.lcf(54301483026227, 54303841038067));
```

Path to compiled lib

Return Type

Parameter Types

Function name from lib

Adding Threading and a Callback

October 4-6, 2017 | Vancouver, BC

```
use std::thread;
use std::sync::Arc;
use std::sync::atomic::{ AtomicUsize, Ordering };
static NTHREADS: i64 = 4;

#[no_mangle]
pub extern fn mt_lcf(n1: i64, n2: i64, done: extern fn(i64)) {
  let lcf = Arc::new(AtomicUsize::new(0));
  let mut children = vec![];
...
```

We need these to create threads and move variables around

Callback parameter

Share this across threads

vec! Is a macro that makes a vector given an array

Adding Threading and a Callback

```
...
for t_number in 0..NTHREADS {
    let thread_lcf = Arc::clone(&lcf);
    children.push(thread::spawn(move || {
      single_lcf_thread(n1, n2, 2 + t_number, NTHREADS, thread_lcf);
    }));
  }
  for child in children {
    let _ = child.join();
  }
  done(lcf.load(Ordering::Relaxed) as i64);
}
```

This lets us give ownership to thread but use a common variable

This isn't a Boolean OR operator, it's a "move" closure

Adding Threading and a Callback

October 4-6, 2017 | Vancouver, BC

Additional parameters

```
fn single_lcf_thread(n1: i64, n2: i64, start: i64, step: i64, lcf: Arc<AtomicUsize>) {
  let mut check = start;
  let stop = if n1 < n2 { n1 + 1 } else { n2 + 1 };
  while check < stop {
    if (n1 % check == 0) && (n2 % check == 0){
      lcf.store(check as usize, Ordering::Relaxed);
    }
    if lcf.load(Ordering::Relaxed) != 0 { return; }
    check += step;
  }
}
```

These lines are the only differences of substance

Calling both versions…

```
const rust = ffi.Library(path.join(__dirname, 'rust-lcf/target/release/liblcf'), {
  lcf: [int64, [ int64, int64 ]],
  mt_lcf: ['void', [ int64, int64, 'pointer' ]],
});

const callback = ffi.Callback('void', [int64], function(factor) {
  console.log(factor);
});

rust.mt_lcf(54301483026227, 54303841038067, callback);
```

Add an entry

Set up the callback

Make the call

So *how fast is fast* exactly?

MT Rust vs Slow JS

Rust FFI is 9.2 times faster

ST Rust vs Fast JS

Rust FFI is 2.1 times faster

MT Rust vs Fast JS

Rust FFI is 7.3 times faster

```
> node lcf-test 54301483026227 54303841038067

100 Iterations of Non-Optimized JS
        Total Time 14.3388571 seconds
        Average Single Iteration: 0.143388571 seconds

100 Iterations of Optimized JS
        Total Time 11.500617541 seconds
        Average Single Iteration: 0.11500617541000001 seconds

100 Iterations of Rust Single Threaded
        Total Time 5.54960728 seconds
        Average Single Iteration: 0.0554960728 seconds

100 Iterations of Rust Multi Threaded
        Total Time 1.565220408 seconds
        Average Single Iteration: 0.01565220408 seconds
```

When should I use Rust?

- You need to do something heavily computational
- You need access to hardware (like running things on the GPU)
- Predictable performance is critical
- Predictable memory footprint is critical

October 4-6, 2017 | Vancouver, BC

What's the down side?

- FFI bridging costs are non-trivial (there are faster options, like Neon or a C++ bridge, but they're harder to use)
- Passing complex data structures can get hairy
- Having a code base in 2 languages is inherently more complex to manage than having just 1.

Interested in learning more?

- A short list to get you started:
- https://github.com/ayasin/rust-talk Slides and code for this talk
- www.rustbyexample.com a great resource to learn Rust in detail
- https://github.com/node-ffi/node-ffi/wiki/Node-FFI-Tutorialhttps://github.com/neon-bindings/neon Neon bindings, which are faster than FFI, but also more complex
- Again, I'm @ayasin in case you haven't had enough of me ☺