Jamstack 与 WebAssembly 在大前端时代的应用与发展

Michael Yuan

WasmEdge Creator and Maintainer

大纲

- 什么是 Jamstack
- Serverless 的崛起
- Wasm 与 Docker
- 用 JavaScript 改进 Wasm 的开发者体验
- 用云原生的工具部署与管理 Wasm

What is Jamstack?

Jamstack is an architecture designed to make the web faster, more secure, and easier to scale. It builds on many of the tools and workflows which developers love, and which bring maximum productivity.

The core principles of pre-rendering, and decoupling, enable sites and applications to be delivered with greater confidence and resilience than ever before.

Pre-rendering

With Jamstack, the entire front end is prebuilt into highly optimized static pages and assets during a build process. This process of pre-rendering results in sites which can be served directly from a CDN, reducing the cost, complexity and risk, of dynamic servers as critical infrastructure.

With so many popular tools for generating sites, like Gatsby, Hugo, Jekyll, Eleventy, NextJS, and very many more, many web developers are already familiar with the tools needed to become productive Jamstack developers.

Enhancing with JavaScript

With the markup and other user interface assets of Jamstack sites served directly from a CDN, they can be delivered very quickly and securely. On this foundation, Jamstack sites can use JavaScript and APIs to talk to backend services, allowing experiences to be enhanced and personalized.

Supercharging with services

The thriving API economy has become a significant enabler for Jamstack sites. The ability to leverage domain experts who offer their products and service via APIs has allowed teams to build far more complex applications than if they were to take on the risk and burden of such capabilities themselves. Now we can outsource things like authentication and identity, payments, content management, data services, search, and much more.

Jamstack sites might utilise such services at build time, and also at run time directly from the browser via JavaScript. And the clean decoupling of these services allows for greater portability and flexibility, as well as significantly reduced risk.

Vercel

微信公众平台 | 小程序

腾讯云 Serverless

netlify

MODERN.JS
现代 Web 工程体系

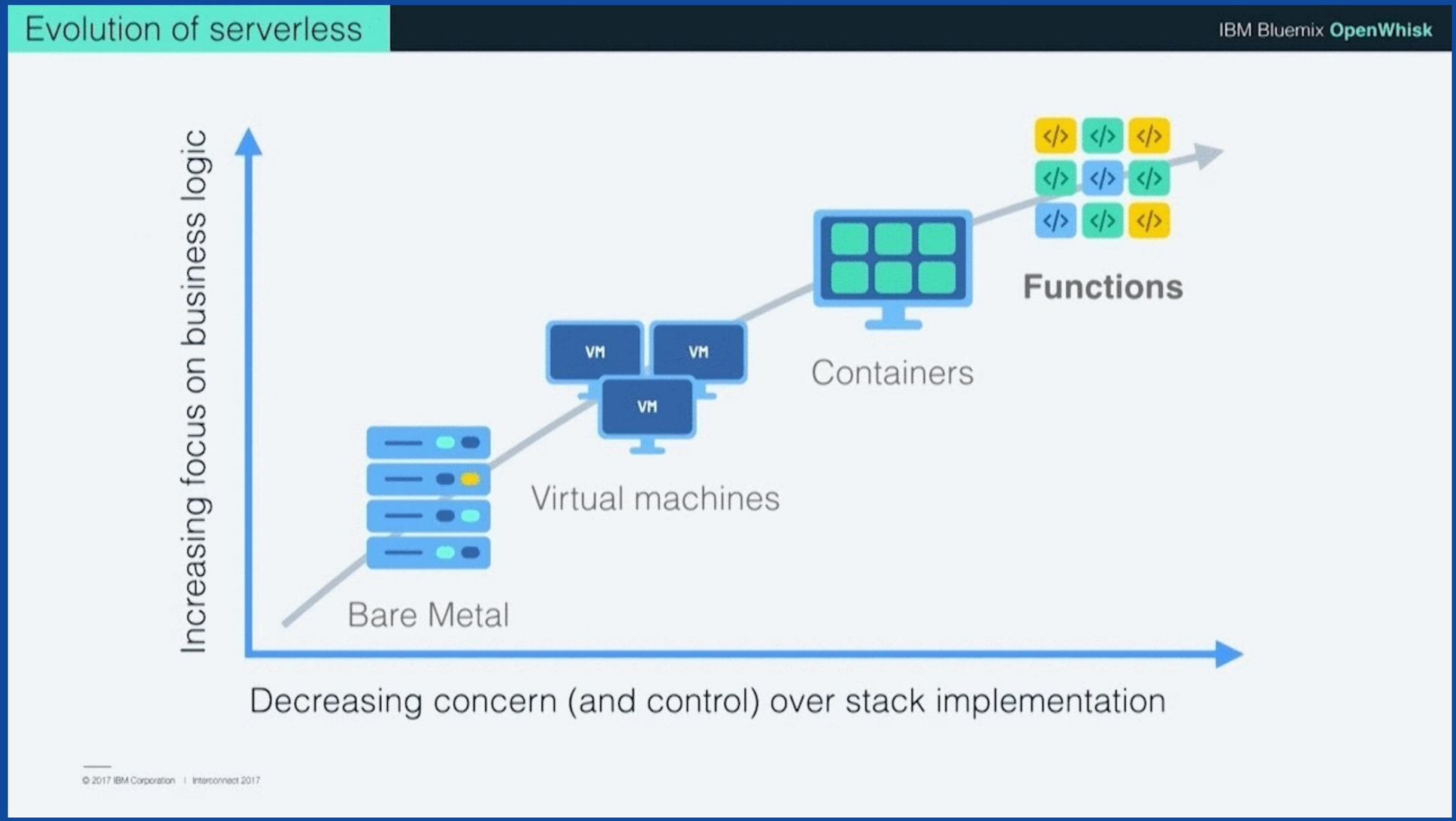
大纲

- 什么是 Jamstack
- Serverless 的崛起
- Wasm 与 Docker
- 用 JavaScript 改进 Wasm 的开发者体验
- 用云原生的工具部署与管理 Wasm

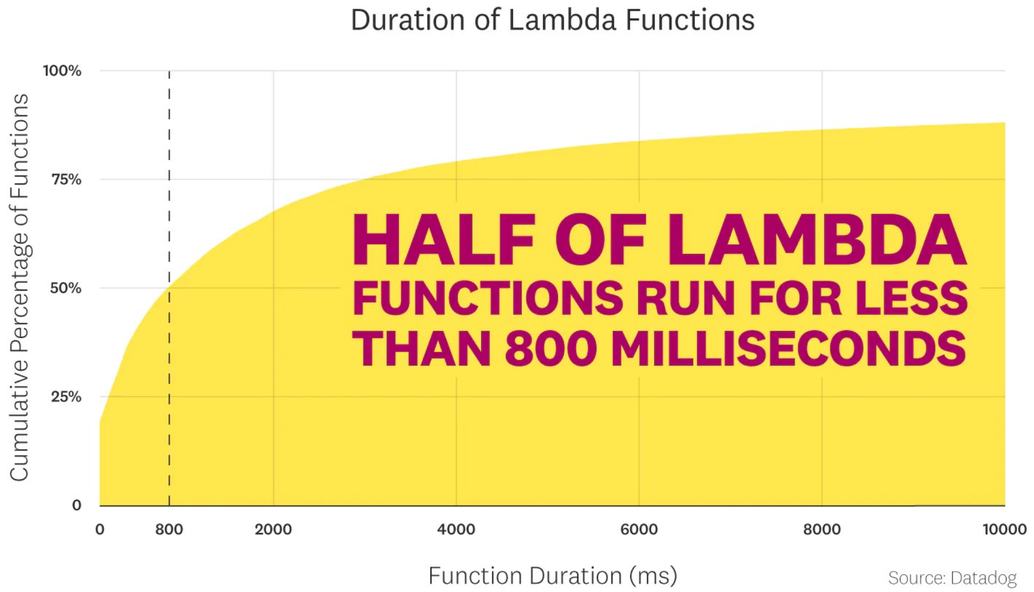
Jamstack

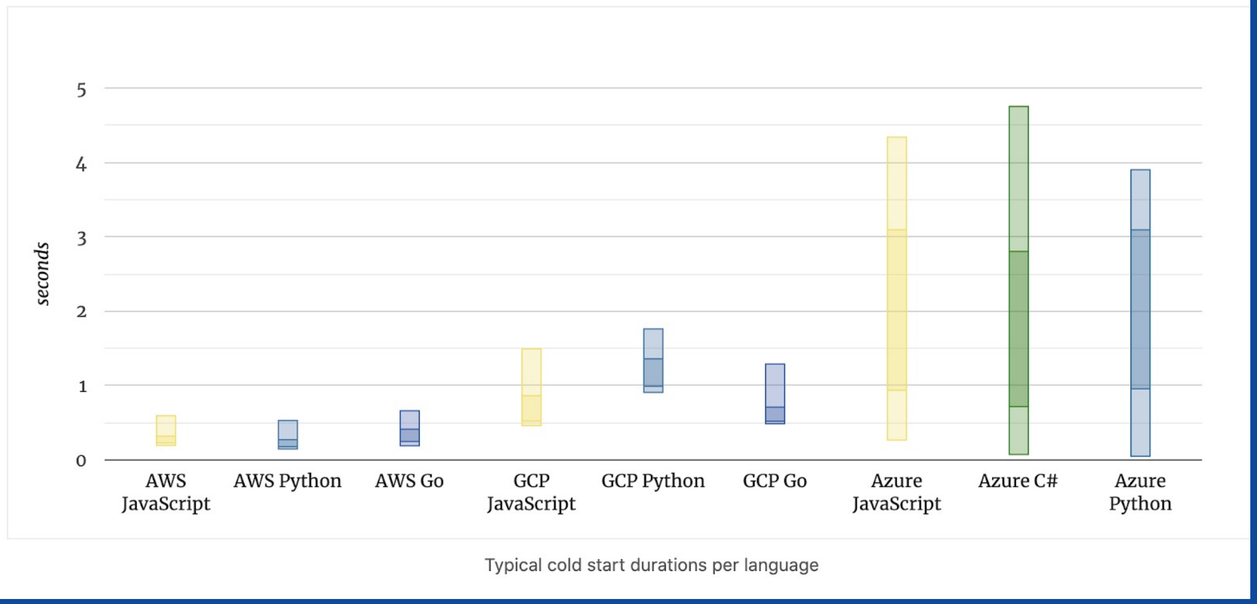
API：微服务或 Serverless 函数

Serverless 的发展



Serverless 函数太慢了



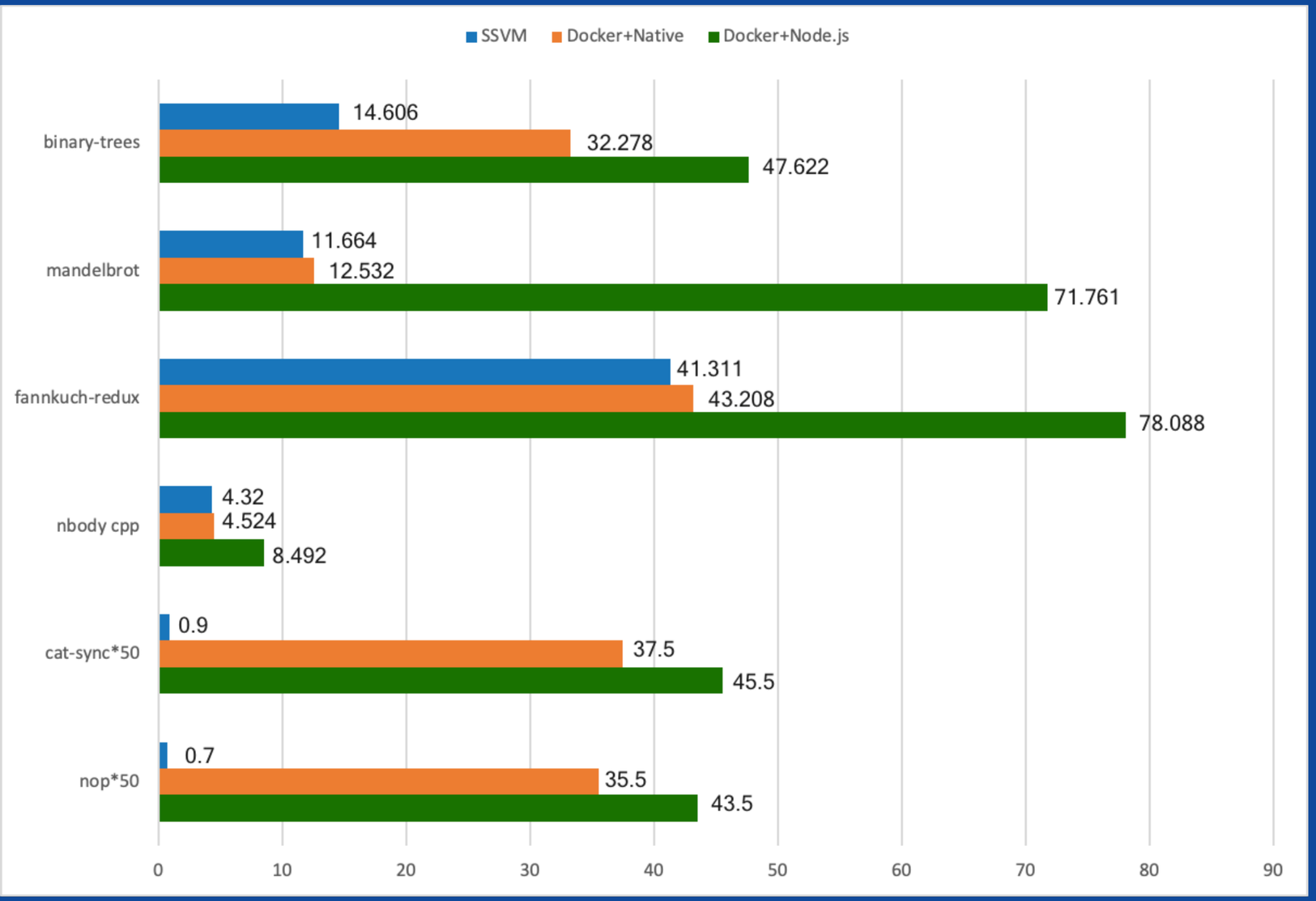


大纲

- 什么是 Jamstack
- Serverless 的崛起
- Wasm 与 Docker
- 用 JavaScript 改进 Wasm 的开发者体验
- 用云原生的工具部署与管理 Wasm

历史总是惊人的相似但是不会简单重复

Wasm 更快更轻



A Lightweight Design for High-performance Serverless Computing, **IEEE Software**, Jan 2021.

https://arxiv.org/abs/2010.07115

Wasm 比 Docker 更抽象

Hypervisor VM 和 microVM	应用容器	高级语言 VM
• AWS Firecracker • 模拟计算机	• Docker • 模拟私有操作系统	• Ruby / Python runtimes、v8、 JVM、 WebAssembly • 模拟一个进程

https://www.infoq.com/news/2020/07/future-serverless-architecture/

https://twitter.com/Joab_Jackson/status/1430634807277629450

目前，Docker 更容易上手

Docker

- 支持任何语言和任何框架
- 经常被当做开发者工具

Wasm

- 虽然多语言，但需要编译器和 SDK 支持
- C/C++、Rust、Swift、Kotlin 和 AssemblyScript 是一等公民
- 被当做 runtime，是操作系统服务的一部分

大纲

- 什么是 Jamstack
- Serverless 的崛起
- Wasm 与 Docker
- 用 JavaScript 改进 Wasm 的开发者体验
- 用云原生的工具部署与管理 Wasm

改进 Wasm 的开发者体验

CLOUD NATIVE
COMPUTING FOUNDATION

WasmEdgeRuntime

为高性能应用优化的

WebAssembly Runtime

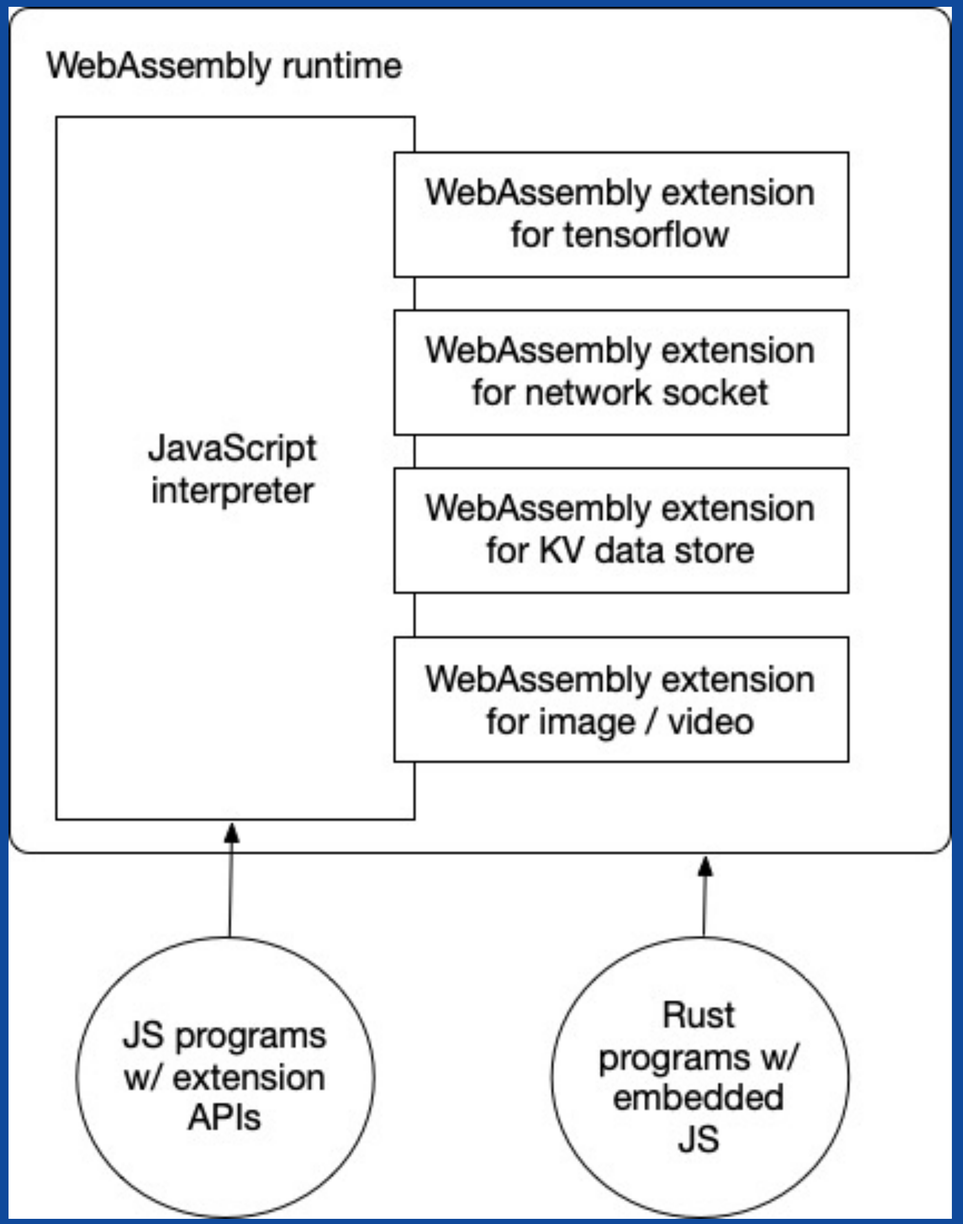
https://github.com/WasmEdge/WasmEdge

重要的用例

- 微服务的云原生 runtime
  - Dapr
  - Kubernetes
  - Service mesh
- 公有云的 serverless runtime
- SaaS 的嵌入式 runtime

Wasm 中的 JavaScript

https://github.com/second-state/wasmedge-quickjs



这是怎么工作的

- 编译成 Wasm 的编译器
  - 支持 ES6 modules
- 把 WasmEdge 扩展做为 JS API
  - Networking
  - 图像处理
  - KV 存储
  - AI 推理
- 允许本机 C 库成为 JS API

使用 TensorFlow 进行AI 推理

33 lines (30 sloc) | 842 Bytes

```
import {TensorflowLiteSession} from 'tensorflow_lite'
import {Image} from 'image'
import * as std from 'std'

let img = new Image('food.jpg')
let img_rgb = img.to_rgb().resize(192,192)
let rgb_pix = img_rgb.pixels()

let session = new TensorflowLiteSession('lite-model_aiy_vision_classifier_food_V1_1.tflite')
session.add_input('input',rgb_pix)
session.run()
let output = session.get_output('MobilenetV1/Predictions/Softmax');
let output_view = new Uint8Array(output)
let max = 0;
let max_idx = 0;
for (var i in output_view){
    let v = output_view[i]
    if(v>max){
        max = v;
        max_idx = i;
    }
}
let label_file = std.open('aiy_food_V1_labelmap.txt','r')
let label = ''
for(var i = 0; i <= max_idx; i++){
    label = label_file.getline()
}
label_file.close()

print('label:')
print(label)
print('confidence:')
print(max/255)
```

异步 HTTP 服务

```
let http_server = new HttpServer('0.0.0.0:8000')
while(true){
    http_server.accept((request)=>{
        let body = request.body
        return {
            status:200,
            headers:{},
            body:'echo:'+body
        }
    });
}
```

ES6 modules

```
function hello(){
    console.log('hello from module_def.js')
}

export {hello}
```

```
export async function hello(){
    console.log('hello from module_def_async.js')
    return "module_def_async.js : return value"
}

export var something = "async thing"
```

```
import { hello as module_def_hello } from './module_def.js'

module_def_hello()

var f = async ()=>{
    let {hello , something} = await import('./module_def_async.js')
    await hello()
    console.log("./module_def_async.js `something` is ",something)
}

f()
```

CommonJS modules

```
print('hello other_module')
module.exports = ['other module exports']
```

```
print('hello one_module');
print('dirname:',__dirname);
let other_module_exports = require('../other_module/main.js')
print('other_module_exports=',other_module_exports)
```

```
import * as one from './one_module/main.js'

print('hello file_module')
```

NPM modules

```
// use ncc build a single file
// $ncc build npm_main.js

import * as std from 'std'

var md5 = require('md5');
console.log(__dirname);
console.log('md5(message)=',md5('message'));
const { sqrt } = require('mathjs')
console.log('sqrt(-4)=',sqrt(-4).toString())

print('write file')
let f = std.open('hello.txt','w')
let x = f.puts("hello wasm")
f.flush()
f.close()
```

用 Rust 来加速 JavaScript

```rust
fn run_rust_function(ctx: &mut Context) {
    println!("\n<----run_rust_function---->");

    struct HelloFn;
    impl JsFn for HelloFn {
        fn call(_ctx: &mut Context, _this_val: JsValue, argv: &[JsValue]) -> JsValue {
            println!("hello from rust");
            println!("argv={:?}", argv);
            JsValue::UnDefined
        }
    }
    let f = ctx.new_function::<HelloFn>("hello");
    ctx.get_global().set("hi", f.into());
    let code = r#"hi(1,2,3)"#;
    let r = ctx.eval_global_str(code);
    println!("return value:{:?}", r);
}
```

这会不会太慢?

- 比 Node + Ubuntu + Docker 快得多
- 对于 Tensorflow 等本机函数，比 V8 快得多
- 不比 V8 解释器慢，但更轻（1/40 大小）
- V8 JIT 不安全

大纲

- 什么是 Jamstack
- Serverless 的崛起
- Wasm 与 Docker
- 用 JavaScript 改进 Wasm 的开发者体验
- 用云原生的工具部署与管理 Wasm

部署与 DevOps

在 Docker 里面部署 Wasm

- Vercel
- Netlify
- 腾讯云 Serverless

Wasm 与 Docker 一起运行

- Dapr
- Kubernetes
- Service mesh

在 Docker 里面部署 Wasm

Vercel

https://www.secondstate.io/articles/vercel-wasmedge-webassembly-rust/

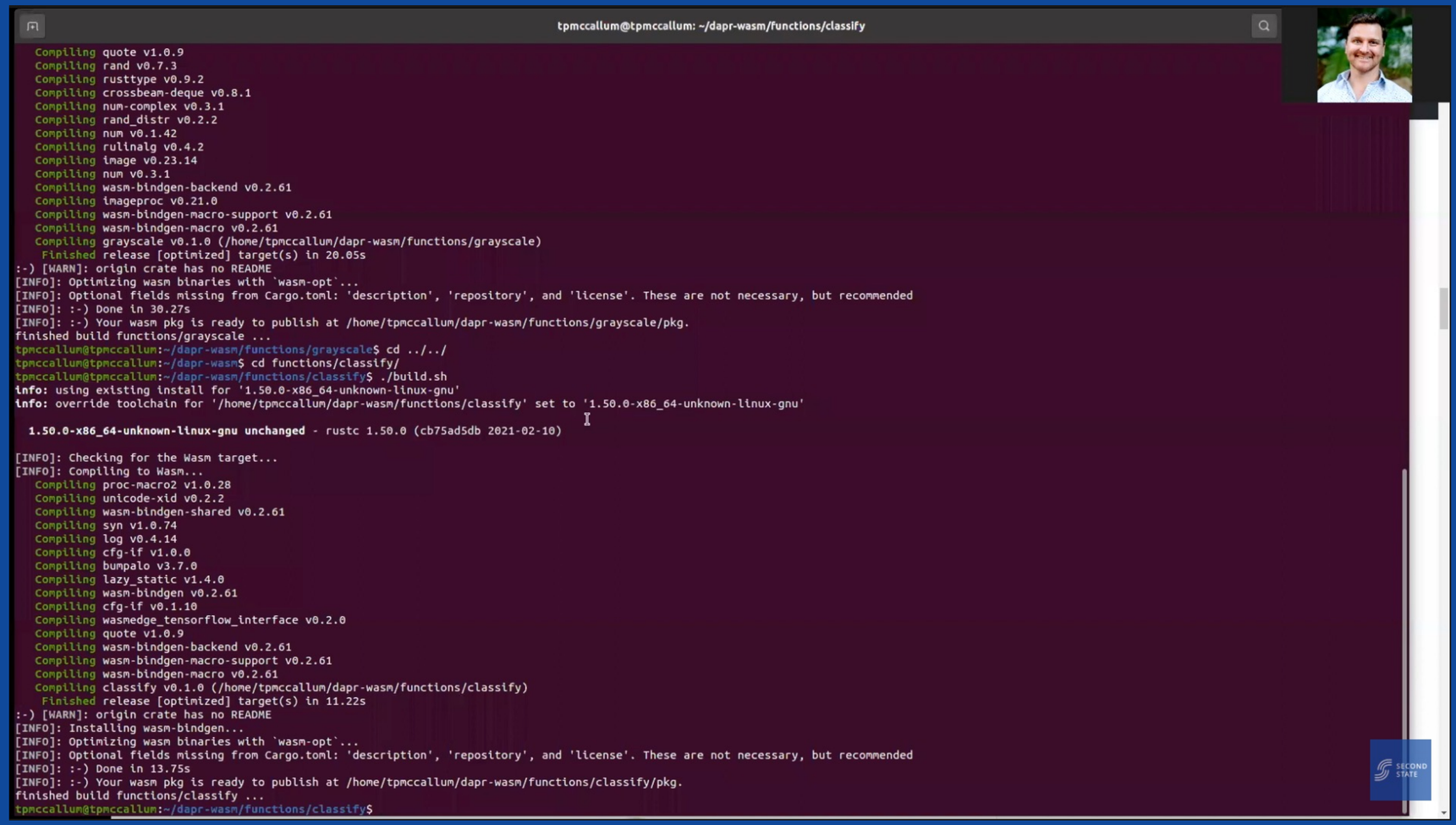
Netlify

https://www.secondstate.io/articles/netlify-wasmedge-webassembly-rust-serverless/

腾讯云 Serverless

https://my.oschina.net/u/4532842/blog/5172639

Dapr 部署

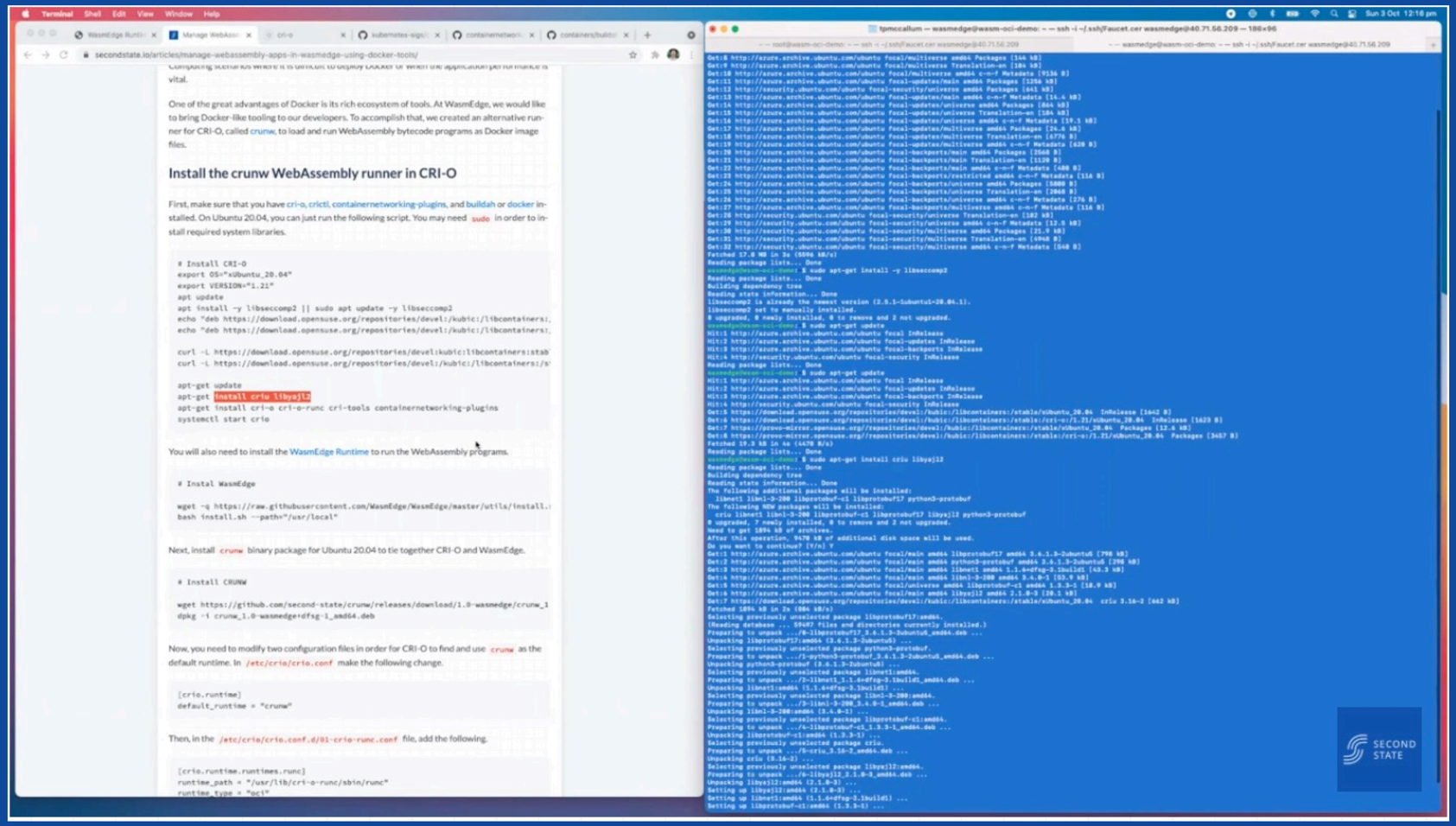


https://github.com/second-state/dapr-wasm

Dapr 上部署的微服务

https://www.secondstate.io/articles/dapr-wasmedge-webassembly/

Kubernetes 上部署



https://www.secondstate.io/articles/manage-webassembly-apps-in-wasmedge-using-docker-tools/

了解更多

https://github.com/WasmEdge/WasmEdge

THANKS

Global

Architect Summit