WebAssembly

Web más alla de JS

_José Madriaza:
_Jose Antonio Castro:
_Benjamín Vicente:

¿Qué es WASM?

WASM...

- Es un formato de **código binario y portátil** (bytecode) para la ejecución en el lado del cliente.
- Es un **lenguaje de destino** de compilación de C, C++, Rust, Go, entre otros.
- Busca habilitar apps de **alto rendimiento en páginas web**.

https://en.wikipedia.org/wiki/WebAssembly

Solución del Problema

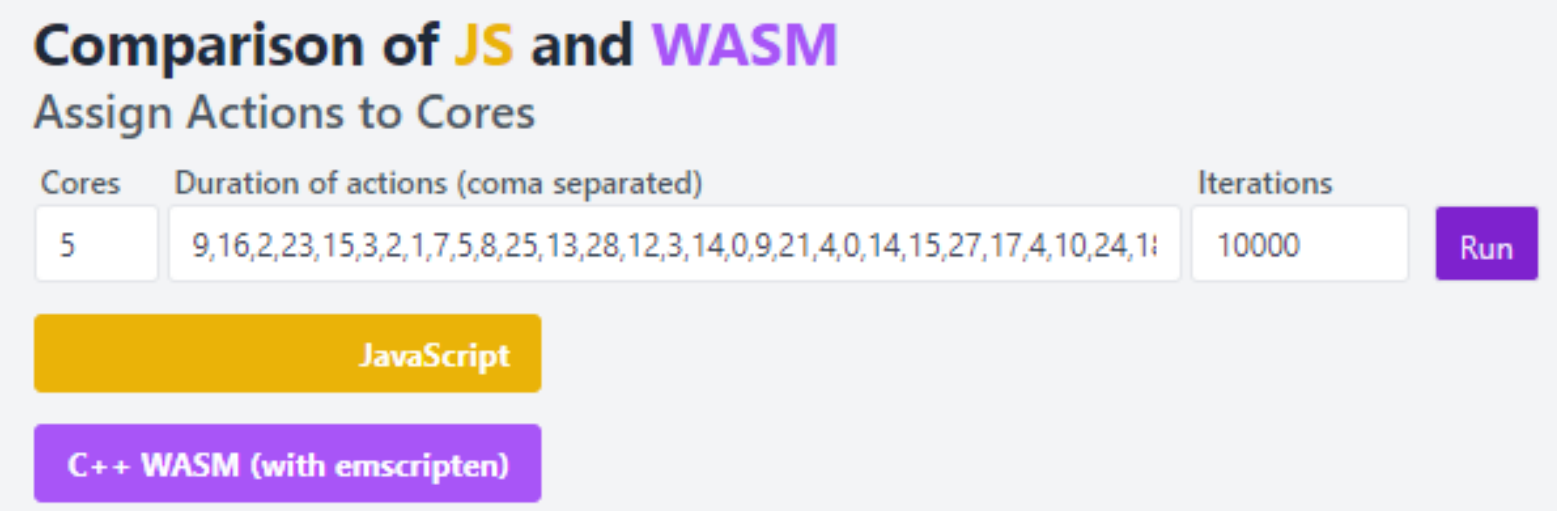
Algoritmo 🤓

```
func asignar_trabajos(m, duraciones):
    tiempos_totales = [0 for _ in range(m)]
    clusters = [[] for _ in range(m)]
    trabajos = sorted(duraciones, reverse=True)

    for trabajo in trabajos:
        idx_min = tiempos_totales.index(min(tiempos_totales))
        tiempos_totales[idx_min] += trabajo
        clusters[idx_min].append(trabajo)

    return max(tiempos_totales)
```

Demo WASM



https://iic3585-2023.github.io/wasm-grupo-01/

Nuestra Experiencia con la Solución

JavaScript: Slow vs Fast

```
jobs.forEach((job) => {
  const minTotalTimeCluster = totalTimes.indexOf(Math.min(...totalTimes));
  clusters[minTotalTimeCluster] = [...clusters[minTotalTimeCluster], job.index];
  totalTimes[minTotalTimeCluster] += job.duration;
});
```

```
for (const job of jobs) {
  const minTotalTimeCluster = totalTimes.indexOf(Math.min(...totalTimes));
  clusters[minTotalTimeCluster].push(job.index);
  totalTimes[minTotalTimeCluster] += job.duration;
}
```

Inmutabilidad disminuye significativamente la velocidad

C++ WASM (emscripten)

```
"cpp-emscripten": async () => {
  const { default: emscripten } = await import("../func/emscripten/scheduler.js");
  const { _assignJobs, _write_vector } = await emscripten();
  return (bins, durations) => {
    for (const duration of durations) _write_vector(duration);
    return _assignJobs(bins, durations.length);
  };
},
```

Codigo: JS importando el código C++ con emscripten

C++ WASM (emscripten)

```cpp
#include <mutex>

std::vector<int> myVector;
std::mutex myVectorMutex;

extern "C" void write_vector(int X) {
  // Lock the mutex before accessing myVector
  std::unique_lock<std::mutex> lock(myVectorMutex);
  myVector.push_back(X);
  // Unlock the mutex after accessing myVector
  lock.unlock();
}
```

Codigo: C++ utilizando vector y mutex

Rust WASM (wasm-pack)

```
"rust-wasm-pack": async () => {
  const { assign_jobs } = await import("../func/wasm-pack/scheduler.js");
  return (bins, durations) => assign_jobs(bins, durations);
},
```

Codigo: JS importando el código Rust

AssemblyScript WASM

- Un TypeScript mucho más quisquilloso

```
class Job {
  duration: i32;
  index: i32;

  constructor(duration: i32, index: i32) {
    this.duration = duration;
    this.index = index;
  }
}
```

```
export function assignJobs(M: i32, durations: i32[]): i32 {
  const totalTimes = new Array<i32>(M).fill(0);

  const clusters = new Array<Array<i32>>(M);
  for (let i: i32 = 0; i < M; i++) clusters[i] = [];

  const jobsUnsorted = durations.map<Job>((duration, index) => new Job(duration, index));
  const jobs = jobsUnsorted.sort((a, b) => b.duration - a.duration);

  for (let i = 0; i < jobs.length; i++) {
    const job = jobs[i];
    const minTotalTimeCluster = findMinIndex(totalTimes);
    clusters[minTotalTimeCluster].push(job.index);
    totalTimes[minTotalTimeCluster] += job.duration;
  }

  const totalTime = findMax(totalTimes);
  return totalTime;
}
```

Go WASM

```
"go-wasm": async () => {
  await import("../func/go-wasm/wasm_exec.js");
  const go = new Go(); ????
  const { default: init } = await import("../func/go-wasm/scheduler.wasm?init");
  const instance = await init(go.importObject);
  go.run(instance);
  return GoAssignJobs; ????
```

Codigo: JavaScript importando GO

Go WASM

```
"go-wasm": async () => {
  await import("../func/go-wasm/wasm_exec.js")
  const go = new Go();
  const { default: init } = await import("../f
  const instance = await init(go.importObject)
  go.run(instance);
  return GoAssignJobs;
```

Codigo: JavaScript importando GO

```
import (
    "fmt"
    "sort"
    "syscall/js"
)

//export GoAssignJobs
func GoAssignJobs(this js.Value, args []js.Value) interface{} {
    M := args[0].Int()
    durations := args[1]
    mappedDurations := make([]int, durations.Length())
    for i := 0; i < durations.Length(); i++ {
        mappedDurations[i] = durations.Index(i).Int()
    }
    return AssignJobs(M, mappedDurations)
}

func main() {
    fmt.Println("[Go] Cargando...")
    done := make(chan struct{}, 0)
    js.Global().Set("GoAssignJobs", js.FuncOf(GoAssignJobs))
    <-done
}
```

??????

- Cual es la sorpresa del más rápido?

JavaScript con mejores EDD

- Cual es la sorpresa del más rápido?
- Mismo algoritmo, pero usa **lista que se mantiene ordenada**.
- El motor de JS tiene **muy optimizado** la función min que hace que **no valga tanto la pena** en algunos casos 😅

Conclusiones

Opiniones y resultados generales

	Velocidad	Facilidad de Uso	Fácil de Aprender	Librerías
JavaScript	x1.3 ~ x3	🤷‍♂️	🤷‍♂️	✅✅
C++ WASM	x1	❌	😬	✅✅
Rust WASM	x1*	✅	😵	✅
AssemblyScript WASM	x4*	✅	🤷‍♂️	😐
Go WASM	x8*	❌❌❌	😐	✅

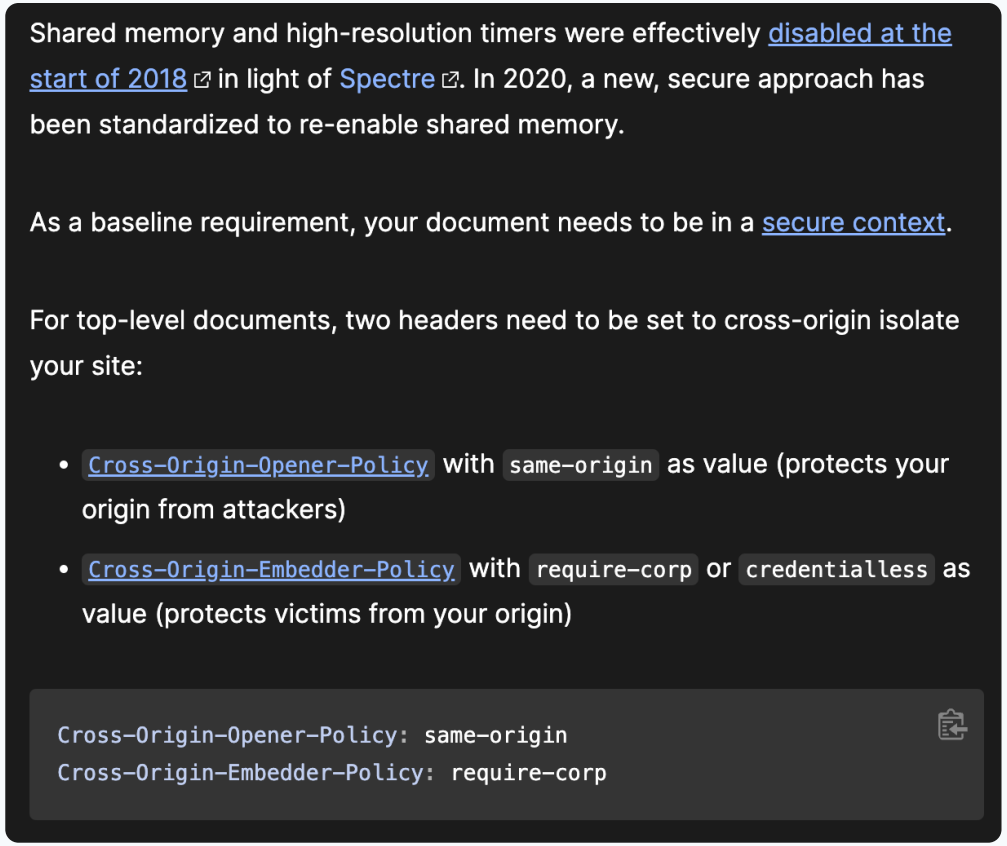
Esto fue con los experimentos realizados.
Pueden existir errores.

Motores de JS son *muy* rápidos

- Crear extructuras de datos para optimizar algoritmos puede **no** valer la pena dado la **velocidad de métodos internos** y la **optimización realizada**.
- En primeras iteraciones puede ser más lento, pero el resultado de las optimizaciones cubren ese tiempo.

Hay que tener ojo con el entorno

APIs como **performance.now** y **SharedArrayBuffer** pueden estar bloqueadas o con comportamiento distinto



https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer#security_requirements

El mayor beneficio es portabilidad

- WASM permite que código de lenguajes establecidos se puedan compilar **sin tener en consideración el entorno**.
- Runtimes como WASMTime permiten **ejecutar código arbitrario** en otros lenguajes de **forma segura**.