Introdução a linguagem de programação Node.js

Node.Js é um ambiente runtime para executar código javascript para construir aplicações de Alta escalabilidade, data-intensive apps e real-time apps.

Node.js é um ambiente runtime JavaScript de código aberto e cross platform. É uma ferramenta popular para quase qualquer tipo de projeto! O Node.js executa o mecanismo JavaScript V8, o núcleo do Google Chrome, fora do navegador. Isso permite que o Node.js tenha muito desempenho.

Um aplicativo Node.js é executado em um único processo, sem criar um novo thread para cada solicitação. O Node.js fornece um conjunto primitivo de operações de E/S assíncronas em sua biblioteca padrão que evita o bloqueio do código JavaScript e, geralmente, as bibliotecas no Node.js são escritas usando paradigmas sem bloqueio, tornando o comportamento de bloqueio a exceção e não a norma.

Informação de mais né? Vamos do começo, o que é um runtime?

Você provavelmente já percebeu que existem muitos runtimes de JavaScript sendo criados e lançados para o mundo hoje em dia. Ferramentas como **Deno, Bun e Cloudflare Workers** surgiram em um curto período de tempo, não era assim no passado.

Um runtime é a mesma coisa que um interpretador?

Interpretador?

Categorias de Linguagens de programação

As linguagens de programação podem ser classificadas como **compiladas ou interpretadas**, dependendo de como o código-fonte é traduzido para código executável.

Linguagens Compiladas:

Em linguagens compiladas, o código-fonte é traduzido integralmente para código de máquina ou código intermediário antes da execução. Esse processo é realizado por um compilador, que gera um arquivo executável que pode ser diretamente executado pelo sistema operacional.

Exemplos de linguagens compiladas:

- C
- C++
- Fortran
- Rust

Vantagens:

- Desempenho otimizado.
- O código-fonte não precisa ser distribuído, apenas o executável.

Desvantagens:

- Necessidade de recompilação para cada plataforma.
- Processo de desenvolvimento pode ser mais lento devido à compilação.

Linguagens Interpretadas:

Em linguagens interpretadas, o código-fonte é traduzido linha por linha durante a execução por um interpretador. Não é gerado um arquivo executável separado, e o código fonte é interpretado e executado diretamente.

Principais características:

1. **Execução:** O código é executado linha por linha pelo interpretador.
2. **Portabilidade:** Geralmente, é mais fácil portar programas entre diferentes sistemas operacionais, pois o interpretador é responsável por adaptar o código.
3. **Desempenho:** Tendem a ser mais lentas, pois não há uma fase de otimização completa antes da execução.

Exemplos de linguagens interpretadas:

- Python
- Ruby
- JavaScript
- PHP

Vantagens:

- Portabilidade entre plataformas.
- Processo de desenvolvimento mais rápido, sem a necessidade de compilação.

Desvantagens:

- Desempenho geralmente inferior.
- Distribuição do código-fonte é necessária, pois não há um executável independente.

E esse tal de runtime?

O termo **"runtime"** refere-se ao período de execução de um programa de computador, ou seja, o tempo durante o qual um programa está sendo executado. Mais especificamente, "runtime" pode ser dividido em duas principais interpretações relacionadas ao contexto de desenvolvimento de software:

1. **Ambiente de Execução (Runtime Environment):**
    - Refere-se ao conjunto de recursos, bibliotecas e serviços necessários para que um programa seja executado. Isso inclui o sistema operacional, bibliotecas de tempo de execução, ambientes virtuais, máquinas virtuais, entre outros. O ambiente de execução fornece suporte para a execução de programas escritos em uma linguagem específica.
2. **Tempo de Execução (Runtime):**
    - Refere-se ao período em que um programa está em execução, desde o momento em que é iniciado até o momento em que é encerrado. Durante o tempo de execução, o programa interage com o ambiente de execução para realizar suas tarefas, acessar recursos do sistema, manipular dados, etc.

Para ilustrar:

- **Exemplo de Ambiente de Execução:**
  - Se você está desenvolvendo em Java, o Java Runtime Environment (JRE) é necessário para executar programas Java. Ele fornece a máquina virtual Java (JVM), bibliotecas e outros recursos necessários para a execução dos programas.
- **Exemplo de Tempo de Execução:**
  - Suponha que você tenha um programa escrito em Python. Quando você o inicia, ele entra no tempo de execução, interage com o ambiente de execução do Python, realiza operações conforme necessário e, finalmente, encerra quando conclui suas tarefas.

Agora que estamos entendidos, vamos voltar a pergunta:

Um runtime é a mesma coisa que um interpretador?

Javascript runtime

No contexto do JavaScript, um runtime do JavaScript é onde uma sequência de código JavaScript é interpretada e avaliada.
Portanto, se você abrir o inspetor do navegador (Chrome, Firefox, etc) digitar um código JavaScript, verá que ele é executado e o resultado é impresso no console. Isso só acontece por causa de um runtime JavaScript sendo executado nos bastidores.

Javascript runtime

Os runtimes JS são comumente vistos em navegadores, mas não se limitam apenas a ambientes Web. Na verdade, você pode criar um programa que use, digamos, o mecanismo V8 do Chrome, e fazer chamadas para ele.

Javascript runtime

Mesmo que você não tenha criado o runtime, este programa também é um runtime JavaScript porque recebe uma string, avalia-a e envia o resultado de volta ao cliente.

Por outro lado, um interpretador é a parte do runtime que irá interpretar esse código e convertê-lo em algo “executável” (ao qual você normalmente não tem acesso).

Javascript runtime

Freqüentemente ouvimos as palavras **runtime** e **engine** como sinônimos. A Engine é o sistema completo que interpreta e valida o código e então o runtime o executa. Para o usuário final, o runtime está embutido na engine e é por isso que vimos pessoas se referindo a eles como a mesma coisa.

Como o Node.js foi construido?

Desde as primeiras versões dessa ferramenta uma coisa que podemos afirmar é a seguinte: O Node.js é na verdade "apenas" um proxy para o mecanismo V8 do Chrome

Não se assuste!

```
Handle<Value> ExecuteString(v8::Handle<v8::String> source, v8::Handle<v8::Value> filename) {
    HandleScope scope;
    TryCatch try_catch;

    Handle<Script> script = Script::Compile(source, filename);
    if (script.IsEmpty()) {
        ReportException(&try_catch);
        ::exit(1);
    }

    Handle<Value> result = script->Run();
    if (result.IsEmpty()) {
        ReportException(&try_catch);
        ::exit(1);
    }
    return scope.Close(result);
}
```

Vamos entender linha por linha:

```
Handle<Value> ExecuteString(v8::Handle<v8::String> source, v8::Handle<v8::Value> filename)
```

Essa linha declara uma função C++ chamada **ExecuteString** que recebe dois parametros:

- **source**: Um identificador para uma string V8 (contendo código JavaScript para execução)
- **filename**: Um identificador para um valor V8 (representando o nome do arquivo de script).

```
HandleScope scope;
```

Esta linha cria um novo escopo de identificador. Na V8, os escopos de identificador são usados para gerenciar o tempo de vida dos identificadores e evitar **vazamentos de memória** (Memory leaks). O escopo garante que todos os identificadores criados nele sejam devidamente limpos quando o escopo for encerrado.

```
TryCatch try_catch;
```

Esta linha cria um objeto TryCatch. **TryCatch** é um mecanismo no V8 para capturar **exceções JavaScript em código C++**. Ele permite interceptar e tratar exceções lançadas durante a execução do script.

```
Handle<Script> script = Script::Compile(source, filename);
```

Esta linha compila o código-fonte JavaScript (fonte) em um script usando o método **Script::Compile**. Ele leva o código-fonte e um nome de arquivo opcional como parâmetros. Se a compilação for bem-sucedida, ele retornará um identificador para o script compilado.

```
if (script.IsEmpty()) {
  ReportException(&try_catch);
  ::exit(1);
}
```

Esta instrução condicional verifica se o identificador de resultado está vazio, indicando que ocorreu uma exceção durante a execução do script.

ReportException(&try_catch) Esta função é chamada novamente para relatar a exceção capturada pelo objeto TryCatch.

::exit(1) Esta linha encerra o programa com um código de saída diferente de zero, indicando uma condição de erro.

```
return scope.Close(result);
```

Esta linha retorna o resultado da execução do script. O método **scope.Close** garante que o identificador de resultado seja fechado corretamente dentro do escopo do identificador atual.

Em essência, o que esse código anterior faz é enviar uma string JS para o V8 e ele executa essa string, retornando um resultado como você pode ver neste trecho de código do Repositório do Node

Curiosidade: Esse commit é de 15 anos atrás feito pelo próprio Ryan Dahl, criador do Node.js e do Deno

Isso vale para os outros runtimes javascript?

Alguns anos atrás (por volta ed 2018) Ryan Dahl apresentou uma talk onde ele dizia de seus arrependimentos sobre o Node.js e apresentou o **Deno**. Um novo Runtime Javascript feito para corrigir todos os problemas que o Node.js tinha.

Deno é escrito em Rust e também é construido em cima do motor V8. O interessante é que os dois seguem o mesmo paradigma.
O trexo de código abaixo foi retirado do repoitório publico do Deno e é basicamente uma variação do código mostrado anteriormente, o programa extende o motor V8 e executa o código JavaScript.

```
deno_core::v8_set_flags(env::args().collect());

let mut js_runtime = create_js_runtime();
let runtime = tokio::runtime::Builder::new_current_thread()
    .enable_all()
    .build()

    .unwrap();
let future = async move {
    js_runtime.execute_script("http_bench_json_ops.js", include_str!("http_bench_json_ops.js"),).unwrap();
    js_runtime.run_event_loop(false).await
};
```

Também temos o **Bun**, mais um Runtime Javascript dessa vez escrito em Zig, que promete ser mais rápido que o Node e o Deno.
Mas no final, você consegue adivinhar o que ele faz? Envia código Javascript para o Runtime Javascript, dessa vez para a engine **JavaScriptCore** também conhecida como SquirrelFish.

Então, qual é a real diferença entre eles? Se eles são **"apenas" proxies** para runtimes que executam código, como pode um ser melhor que o outro?
A resposta não depende do JS em si, mas da maneira como eles controlam o fluxo de dados do motor/engine e como eles lidam com as tarefas do sistema operacional.

Entendendo os componentes principais utilizados no Node.js

Node.js é um sistema integrado em três componentes principais:

- Motor JavaScript V8 do Chrome
- Libuv - Operações assíncronas
- Camada C++ - Funções para ajudar a controlar o fluxo de dados

Componentes principais: 1-3 - Motor V8

Como dito anteriormente, o motor V8 é responsável por interpretar o código JavaScript e executa-lo. Porém ele faz muito mais do que isso.
Ele traduz seu código JavaScript para instâncias de objetos C++
Dessa forma temos um fato interessante, a função console.log não existe no V8.
Caso chame essa função você receberá um erro: **"console is undefined"**.

console.log não é JavaScript... é uma função C++ que chama a função **printf**

Isso ocorre porque o V8 executa apenas o que está na especificação do ECMAScript, como promessas, classes, funções e variáveis, mas outras funções comumente usadas como console, setTimeout e setInterval não fazem parte desta especificação, então o V8 nem sabe o que são.

Abaixo está um trexo de código que o autor criou para implementar seu próprio `console.log`. Ele deu a sua função o nome de Print.

```
void Print(const v8::FunctionCallbackInfo<v8::Value> &args) {
  bool first = true;
  for (int i = 0; i < args.Length(); i++) {
    v8::HandleScope handle_scope(args.GetIsolate());
    if (first) {
      first = false;
    }
    else {
      printf(" ");
    }
    v8::String::Utf8Value str(args.GetIsolate(), args[i]);
    printf("%s", *str);
  }
  printf("\n");
  fflush(stdout);
}
```

Verifique o código completo

```
void Print(const v8::FunctionCallbackInfo<v8::Value> &args)
```

Esta linha declara uma função C++ chamada Print que usa um único parâmetro do tipo `v8::FunctionCallbackInfo<v8::Value>`. Este parâmetro representa os argumentos passados para a função quando ela é chamada de JavaScript.

```
bool first = true;
```

Esta linha declara uma variável booleana nomeada first e a inicializa como true. Esta variável é usada para controlar se este é o primeiro argumento a ser impresso.

```
for (int i = 0; i < args.Length(); i++)
```

Esta linha inicia um loop que itera sobre os argumentos passados para a função. `args.Length()` retorna o número de argumentos passados.

```
v8::HandleScope handle_scope(args.GetIsolate());
```

Esta linha cria um identificador de escopo. Os identificadores de escopo são usados no V8 para gerenciar o tempo de vida dos identificadores e evitar vazamentos de memória. O escopo é inicializado com o isolate associado aos argumentos.

```
if (first) {
  first = false;
}
else {
  printf(" ");
}
```

Esta instrução condicional verifica se este é o primeiro argumento a ser impresso. Se for o primeiro argumento, first será definido como false. Caso contrário, será impresso um espaço para separar os argumentos.

```
v8::String::Utf8Value str(args.GetIsolate(), args[i]);
```

Esta linha converte o i-ésimo argumento em uma string codificada em UTF-8. Ele usa a classe Utf8Value fornecida pela V8 para realizar esta conversão.

```
printf("%s", *str);
```

Esta linha imprime a string codificada em UTF-8 na saída padrão usando printf. *str desreferencia o ponteiro para obter o valor real da string.

```c
printf("\n");
fflush(stdout);
```

`printf("\n");` Esta linha imprime um caractere de nova linha na saída padrão, garantindo que a saída subsequente comece em uma nova linha.

`fflush(stdout);` Esta linha libera o fluxo de saída padrão, garantindo que qualquer saída em buffer seja gravada no console imediatamente.

Não fique muito preso na implementação do C++, no final, a resposta é o método printf, uma das funções C++ mais primitivas também é usada no Node.js para imprimir o código na saída padrão.

Depois de criar a função Print, é preciso injetá-la no contexto Global do JavaScript.

```
v8::Local<v8::Context> CreateContext(v8::Isolate *isolate) {
  // Cria um modelo para o objeto global.
  v8::Local<v8::ObjectTemplate> global =
  v8::ObjectTemplate::New(isolate);
  // Vincula a função global 'print' ao callback C++ Print.
  global->Set(isolate, "print", v8::FunctionTemplate::New(isolate, Print));
  // Cria um novo contexto.
  return v8::Context::New(isolate, NULL, global);
}
```

Em global->Set é onde mapeamos a função C++ Print para se tornar uma função de print disponível no contexto Global do JavaScript.

Então podemos utiliza-la desta forma no arquivo ***index.js***:

```
print("Hello World");
```

Ao usar um arquivo JavaScript chamando a função de print que acabou de ser implementada em C++, a engine (V8) irá entendê-lo como uma função JavaScript e chamar a função construída a partir de C++.

É por isso que o autor informa que o Node.js é uma extensão do motor V8 porque você pode implementar quaisquer funções que o JavaScript não tenha, da mesma forma que foi feita anteriormente.
Módulos como crypto, HTTP, net, e child process também foram construídos desta forma, eles são apenas funções em C++ que extendem o comportamento padrão da V8.
É assim que Deno e Bun podem fazer suas implementações de forma diferente e afirmarem ser mais rápidos que o Node.js

JavaScript Promises

Outra parte muito importante do nosso runtime são as promessas. Como você talvez não saiba, as promessas são, na verdade, um padrão de projeto, não são funções assíncronas. O que elas fazem é agrupar funções assíncronas e retornar um objeto que controla se uma função foi concluída com êxito ou não.

Um pouco sobre como o Node.js funciona

How Node.js Works | Mosh

Node deve ser utilizado para aplicações intensiva de data IO e também para aplicações real time

Não usar node para: Processamento de imagens e vídeos | CPU-intensive apps

Essas operações intensivas de CPU geralmente são demoradas (O que eventualmente vai causar o bloqueio do Event Loop)

Como as promessas fazem parte da especificação ECMAScript, você pode executá-las diretamente na V8.
Por enquanto, observe apenas que o **timeout**.
Estamos apenas envolvendo-o em um objeto Promise. Quando a camada C++ chama a função de callback da função timeout, ela também chama a função **resolve do objeto Promise**, que informa à nossa palavra-chave **await** que a função foi concluída com sucesso.

```
const setTimeout = (ms, cb) => timeout(ms, 0, cb);
const setTimeoutAsync = (ms) =>
  new Promise((resolve) => setTimeout(ms, resolve));

(async function asyncFn() {
  print(new Date().toISOString(), "waiting a sec...");
  await setTimeoutAsync(1000);
  print(new Date().toISOString(), "waiting a sec...");
  await setTimeoutAsync(1000);
  print(new Date().toISOString(), "finished at");
})();
```

Este código demonstra o uso de operações assíncronas em JavaScript, particularmente utilizando setTimeout para introduzir atrasos na execução e Promise para lidar com operações assíncronas.

Vamos detalhar o código passo a passo:

```
const setTimeout = (ms, cb) => tempo limite (ms, 0, cb);
```

Esta linha define uma função **setTimeout** que recebe dois parâmetros: **ms**(milissegundos) e **cb** (função de callback).

```
const setTimeoutAsync = (ms) =>
  new Promise((resolver) => setTimeout(ms, resolver));
```

Esta linha define uma função **setTimeoutAsync** que leva um parâmetro ms (milissegundos).
Dentro desta função, ela cria e retorna uma **new Promise**.

O **construtor Promise** usa uma função com resolução como parâmetro. Dentro desta função, **setTimeout** é chamado com os milissegundos (ms) fornecidos e a função resolve como seu callback.

Isso significa que após **ms** milissegundos, a promessa será resolvida.

- Um objeto Promise é apenas uma maneira bonita de lidar com funções de callback em uma única linha.

```
(async function asyncFn() { ... })();
```

Esta é uma expressão de função assíncrona invocada imediatamente (Immediately invoked asynchronous function expression / IIFE). Ela define uma função assíncrona chamada **asyncFn** e a invoca imediatamente.

```
print(new Date().toISOString(), "waiting a sec...");
```

Esta linha registra a data e hora atuais no formato ISO junto com a mensagem "waiting a sec...".

```
await setTimeoutAsync(1000);
```

Esta linha pausa a execução da função assíncrona até que a promessa retornada por **setTimeoutAsync** seja resolvida (ou seja, após 1000 milissegundos).
Durante esse tempo, o fluxo de controle é retornado ao loop de eventos, permitindo a execução de outras tarefas.
A próxima linha é semelhante à anterior, aguardando mais 1000 milissegundos antes de prosseguir.

```
print(new Date().toISOString(), "finished at");
```

Esta linha registra a data e hora atuais junto com a mensagem "finished at", indicando a conclusão da operação assíncrona.

No geral, o código demonstra como usar **funções assíncronas, Promises e setTimeout** para introduzir atrasos na execução do código JavaScript sem bloquear o thread principal.