



Search for:

NEWSLETTER

Name

Email address:

Your email address

Sign up

RECENT POSTS

- Local Docker images no Minikube
- Node.js Design Patterns: Singleton
- Node.js | 3 Maneiras de lidar com serviços externos em testes de integração
- Propriedades publicas e privadas chegaram no Node.js v12 com o V8 7.2
- Node.js V12 released 🚀🌟O que vem de novo por aí?

RECENT COMMENTS

- Fellipe Abreu on [Construindo uma API testável com Node.js](#)
- Waldemar Neto on [Suportando módulos EcmaScript \(ESM\) no Node.js sem Babel](#)
- Vitor Luiz Cavalcanti on [Suportando módulos EcmaScript \(ESM\) no Node.js sem Babel](#)
- Waldemar Neto on [Dockerizando aplicações Node.js + npm 5](#)
- jacksonfdam on [Dockerizando aplicações Node.js + npm 5](#)

ARCHIVES

- July 2019
- May 2019
- April 2019
- February 2019
- July 2018
- May 2018
- April 2018
- March 2018
- February 2018
- October 2017
- September 2017
- August 2017
- April 2017
- February 2017
- November 2016
- October 2016
- September 2016
- August 2016
- July 2016
- June 2016
- May 2016
- March 2016
- February 2016
- January 2016
- October 2015
- September 2015
- August 2015
- July 2015
- June 2015
- May 2015
- April 2015
- March 2015
- February 2015
- January 2015
- December 2014
- November 2014
- July 2014

CATEGORIES

- About the Book
- CakePHP
- Design Patterns
- DevOps
- Docker
- ElasticSearch
- General Development
- Git
- Java
- Javascript
- kubernetes
- Node.js
- OAuth
- OpenCV
- PHP
- RESTful APIs
- Symfony2
- Uncategorized

- E-mail
- @waldemarnt
- Facebook
- GooglePlus
- LinkedIn
- GitHub
- RSS

- HOME
- SOBRE MIM
- CANAL DO YOUTUBE
- EU NO IMASTERS
- GITHUB
- CONTATO



Node.js: V8, Single thread e I/O não bloqueante.

NOV 27, 2016

|

2

COMMENTS

Share this content

Like

0

Shares

Escrevendo o meu livro **Construindo APIs testáveis com Node.js** acabei fazendo uma imersão no código do *google v8* e também no *Node.js* para entender como eles trabalham juntos, agora resolvi dividir esse aprendizado com vocês, esse conteúdo também estará no livro, então todo o *feedback* é muito bem vindo.

O Google V8

O *V8* é uma *engine* criada pela *Google* para ser usada no *browser chrome*. Em 2008 a *Google* tornou o *V8 open source* e passou a chamá-lo de *Chromium project*. Essa mudança possibilitou que a comunidade entendesse a *engine* em si, além de compreender como o *javascript* é interpretado e *compilado* por esta.

O *javascript* é uma linguagem interpretada, o que o coloca em desvantagem quando comparado com linguagens *compiladas*, pois cada linha de código precisa ser interpretada enquanto o código é executado. O *V8 compila* o código para linguagem de máquina, além de otimizar drasticamente a execução usando heurísticas, permitindo que a execução seja feita em cima do código compilado e não interpretado.

O Node.js é single thread

A primeira vista o modelo *single thread* parece não fazer sentido, qual seria a vantagem de limitar a execução da aplicação em somente uma *thread*? Linguagens como *Java*, *PHP* e *Ruby* seguem um modelo onde cada nova requisição roda em uma *thread* separada do sistema operacional. Esse modelo é eficiente mas tem um custo de recursos muito alto, nem sempre é necessário todo o recurso computacional aplicado para executar uma nova *thread*.

O *Node.js* foi criado para solucionar esse problema, usar programação assíncrona e recursos compartilhados para tirar maior proveito de uma *thread*.

O cenário mais comum é um servidor *web* que recebe milhões de requisições por segundo; Se o servidor iniciar uma nova *thread* para cada requisição vai gerar um alto custo de recursos e cada vez mais será necessário adicionar novos servidores para suportar a demanda. O modelo assíncrono *single thread* consegue processar mais requisições concorrentes do que o exemplo anterior, com um número bem menor de recursos.

Ser *single thread* não significa que o *Node.js* não usa *threads* internamente, para entender mais sobre essa parte devemos primeiro entender o conceito de *I/O* assíncrono não bloqueante.

I/O assíncrono não bloqueante

Essa talvez seja a característica mais poderosa do *Node.js*, trabalhar de forma não bloqueante facilita a execução paralela e o aproveitamento de recursos.

Para entender melhor, vamos pensar em um exemplo comum do dia a dia. Imagine que temos uma função que realiza várias ações, como por exemplo: uma operação matemática, ler um arquivo de disco, e transformar o resultado em uma *String*. Em linguagens bloqueantes como *PHP*, *Ruby* e etc, cada ação será executada apenas depois que a ação anterior for encerrada, no exemplo citado a ação de transformar a *String* terá que esperar uma ação de ler um arquivo de disco, que pode ser uma operação pesada, certo?

Vamos ver um exemplo de forma síncrona, ou seja, bloqueante:

```
1  const fs = require('fs');
2  let fileContent;
3  const someMath = 1+1;
4
5
6
7  try {
8    fileContent = fs.readFileSync('big-file.txt', 'utf-8');
9    console.log('file has been read');
10 } catch (err) {
11   console.log(err);
12 }
13
14 const text = `The sum is ${ someMath }`;
15 console.log(text);
```

Nesse exemplo, a última linha de código com o *console.log* terá que esperar a função *readFileSync* do module de *file system* executar, mesmo não possuindo ligação alguma com o resultado da leitura do arquivo.

Esse é o problema que o *Node.js* se propôs a resolver, possibilitar que ações não dependentes entre si sejam desbloqueadas. Para solucionar isso o *Node.js* depende de uma funcionalidade chamada *high order functions* que basicamente possibilitam passar uma função por parâmetro para outra função, assim como uma variável, as funções passadas como parâmetro serão executadas posteriormente, como no exemplo a seguir:

```
1  const fs = require('fs');
2
3
4  const someMatch = 1+1;
5
6
7  fs.readFile('big-file.txt', 'utf-8', function (err, content) {
8    if (err) {
9      return console.log(err)
10    }
11
12    console.log(content)
13  })
14
15
16
17  const text = `The response is ${ someMatch }`;
18  console.log(text);
```

No exemplo acima usamos a função *readFile* do módulo *file system*, assíncrona por padrão. Para que seja possível executar alguma ação quando a função terminar de ler o arquivo é necessário passar uma função por parâmetro, essa função será chamada automaticamente quando a função *readFile* finalizar a leitura.

Funções passadas por parâmetro para serem chamadas quando a ação é finalizada são chamadas de *callbacks*. No exemplo acima o *callback* recebe dois parâmetros injetados automaticamente pelo *readFile*: *err*, que em caso de erro na execução irá possibilitar o tratamento do erro dentro do *callback*, e *content* que é a resposta da leitura do arquivo.

Para entender como o *Node.js* faz para ter sucesso com o modelo assíncrono é necessário entender também o *Event Loop*.

Share this content

Like

0

Shares

- javascript
- node single thread
- node v8
- nodejs
- nodejs io assincrono

2 Comments

waldeco

1

Login

- Recommend
- 1
- Tweet
- Share

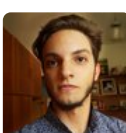
Sort by Newest

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS

Name



Dayman Novaes

 • 3 years ago

Waldemar, gostei do seu artigo, é simples e introduz de forma clara três conceitos importantes pra entender o Javascript. Encontrei apenas um erro de português, que é usar "mesmo" para retornar termos, apesar do uso comum, é incorreto. Na frase "além de compreender como o javascript é interpretado e compilado pela mesma." poderia-se substituir "pela mesma" por "por esta", por exemplo.

Abraços!

1

^

 |

^

 • Reply • Share

Waldemar Neto

Mod

 ➔ Dayman Novaes • 3 years ago

@Dayman Novaes muito obrigado pelo feedback, pesquisei e realmente faz sentido. Como vai para o livro, esse tipo de feedback é muito útil pra mim. Grande abraço

2

^

 |

^

 • Reply • Share

ALSO ON WALDECO

Dockerizando aplicações Node.js + npm 5

2

comments

 • 2 years ago

jacksonfdam — Complementando, em produção sempre usamos separado os databases, em seus devidos containers. Seguem umas sugestões:<https://gist.github.com/jac...>

Docker!! 5 dicas para otimizar seu Dockerfile – Série DevOps

1

comment

 • 2 years ago

Diogo Alves Miranda Barbosa — Ótimas dicas, realmente valiosas, parabéns!!!

Javascript Hoisting o que é?

2

comments

 • 4 years ago

Rafael Fidelis — Hoisting é uma das paradas mais ocultas da linguagem, é muito interessante aprender e conseguir observar isso no dia a dia!

Dependências consistentes no NPM com NPM Shrinkwrap

5

comments

 • 3 years ago

Natan — Muito show seu artigo abraço

- Subscribe
- Add Disqus to your siteAdd DisqusAdd
- Disqus' Privacy PolicyPrivacy PolicyPrivacy