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

NOV 27, 2016 | 2 COMMENTS

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:

```
const fs = require('fs');
let fileContent;
const someMath = 1+1;


try {
  fileContent = fs.readFileSync('big-file.txt', 'utf-8');
  console.log('file has been read');
} catch (err) {
  console.log(err);
}


const text = `The sum is ${ someMath }`;
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:

```
const fs = require('fs');


const someMatch = 1+1;


fs.readFile('big-file.txt', 'utf-8', function (err, content) {
 if (err) {
 return console.log(err)
 }


 console.log(content)
})


const text = `The response is ${ someMatch }`;
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*.

javascript *node single thread* *node v8* *nodejs* *nodejs io assíncrono*