
HELLO
WORLD


JAVASCRIPT
ASSÍNCRONO

OPERAÇÕES SÍNCRONAS

Em JavaScript, uma operação síncrona é aquela que é executada de maneira sequencial. Isso significa que o código é executado linha por linha, e uma linha não pode ser executada até que a anterior seja concluída.

```
console.log("Primeira linha");
console.log("Segunda linha");
console.log("Terceira linha");
```

```
Primeira linha
Segunda linha
Terceira linha
```

OPERAÇÕES ASSÍNCRONAS

Uma operação assíncrona é aquela que permite que outras operações sejam executadas enquanto se espera por uma tarefa específica ser concluída. Isso é particularmente útil para tarefas demoradas, como chamadas de rede ou leitura de arquivos.

```
console.log("Primeira linha");

setTimeout(() => {
    console.log("Segunda linha (após 2 segundos)");
}, 2000);

console.log("Terceira linha");
```

```
Primeira linha
Terceira linha
Segunda linha (após 2 segundos)
```

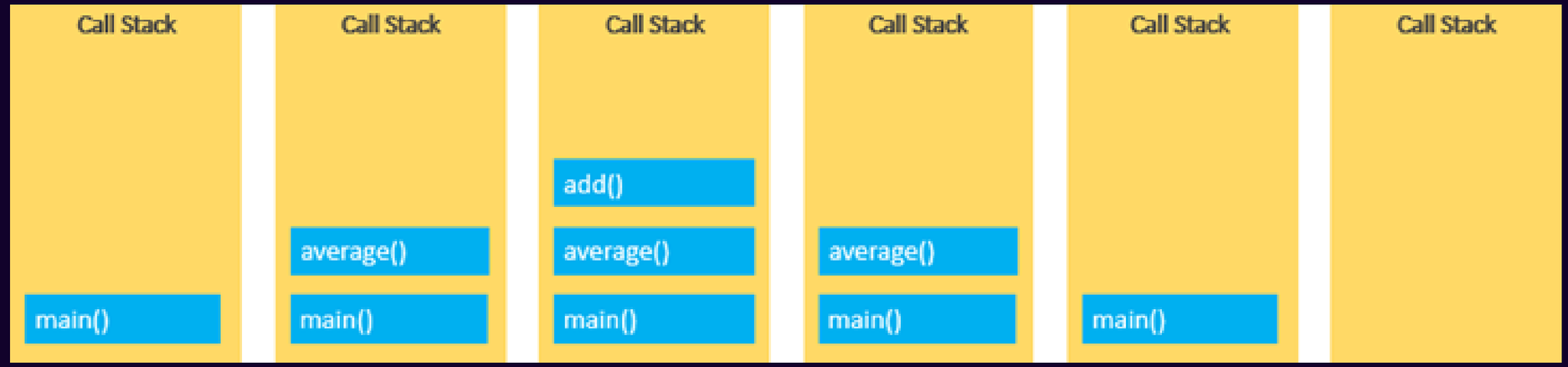

COMO O
JAVASCRIPT
FUNCIONA

SINGLE-THREAD

Node.js é uma plataforma orientada a eventos que opera sob o princípio de uma única thread, gerenciando eficientemente a pilha de eventos ou Call Stack. Esta pilha adota a estratégia LIFO (Last In, First Out), onde a última entrada é a primeira a ser processada. As operações de fundo, ou background, são administradas por 'workers' que operam em segundo plano, e estes podem executar tarefas multi-thread.

Os 'workers' são, essencialmente, processos de I/O assíncronos e não-bloqueantes, gerenciados pela libuv. Esta é uma biblioteca open-source, multiplataforma, escrita em C, que se vale de um pool de threads para gerir operações paralelas. A abordagem de thread única na gestão da Call Stack é vital para o alto desempenho do Node.js.

```
function add(a, b) {
  return a + b;
}
function average(a, b) {
  return add(a, b) / 2;
}

let x = average(10, 20);
```



STACK OVERFLOW E RANGEERROR

Também é importante notar que a Call Stack é uma estrutura de dados limitada em tamanho. Se seu aplicativo tiver muitas funções aninhadas ou recursivas, ele pode consumir espaço rapidamente na Call Stack e causar erros. Se o número de contextos de execução exceder o tamanho da pilha, ocorrerá um erro de estouro de pilha.

Por exemplo, quando você executa uma função recursiva que não tem uma condição de saída, o mecanismo JavaScript gerará um erro de estouro de pilha:

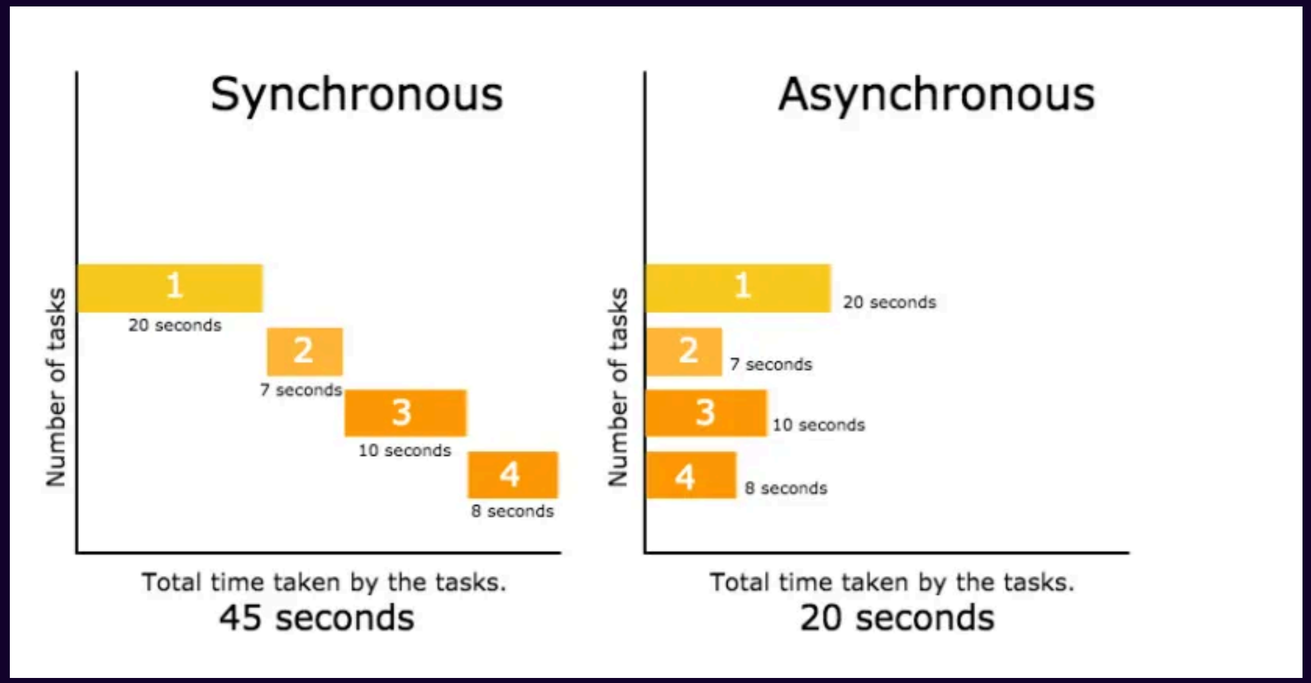
```
função fn() {
  fn();
}

fn(); // estouro de pilha
```

O que causa esses erros na pilha de chamadas?

- Problemas no tratamento de recursão.
- Se você tiver uma função que se chama várias vezes, vários quadros de pilha serão gerados na Pilha de Chamadas, o que pode causar um estouro de pilha e fazer com que seu aplicativo trave.
- Operações fora do alcance.

CONCORRÊNCIA

Apesar de ser single-threaded, JavaScript consegue lidar com várias tarefas simultaneamente (concorrência) usando a fila de eventos (event loop). Operações assíncronas são enviadas para a fila de eventos, e quando o thread principal está livre, essas operações são executadas.



Javascript assíncrono: Em um ambiente assíncrono, ou não bloqueante, o código pode ser executado imediatamente em vez de esperar o anterior terminar. Ter um código que roda assincronamente permite que um usuário busque dados do servidor, execute uma função com atraso e lide com múltiplas requisições simultaneamente. Isso ajuda a reduzir o tempo de espera para o usuário.

Os gatilhos mais comuns para operações Javascript assíncronas são:

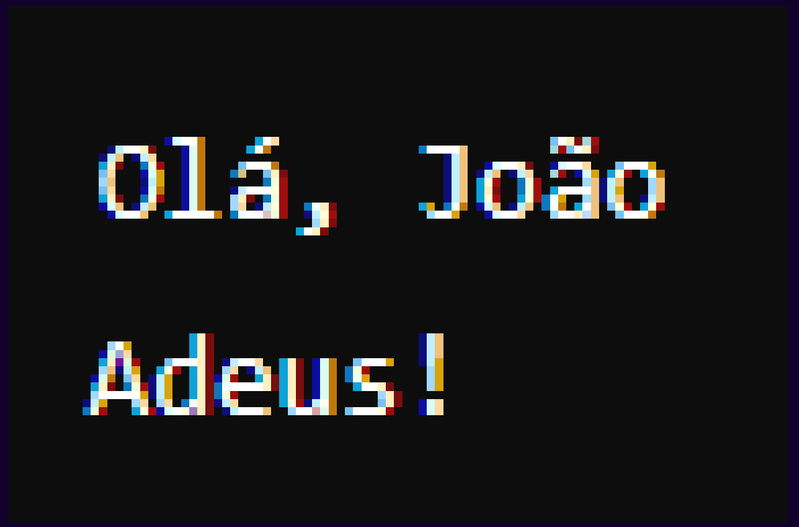
1. Loop de eventos
2. Ligar de volta
3. Promessas
4. Assíncrono/Aguardando


CALLBACKS

O QUE É UM CALLBACK

Em JavaScript, callback é uma função que pode ser passada como argumento para outra função. A função de chamada pode então executar a função de callback mais tarde, normalmente quando uma operação assíncrona tiver sido concluída.

```
function saudacao(nome, callback) {
    console.log(`Olá, ${nome}`);
    callback();
}

function despedida() {
    console.log("Adeus!");
}

saudacao("João", despedida);
```



Callbacks são usados para manipular operações assíncronas em JavaScript porque a linguagem não tem suporte interno para threads ou simultaneidade. Operações assíncronas são operações que não bloqueiam o thread principal de execução, como buscar dados de um servidor ou gravar em um arquivo.

Obtendo dados de uma API:

```
function  fetchData ( url, retorno de chamada ) {
  fetch (url)
    . then ( res => res. json ())
    . then ( data =>  retorno de chamada ( null , data))
    . catch ( err =>  retorno de chamada (err, null ));
}

// Função de retorno de chamada
function  handleData ( err, data ) {
  if (err) {
    console . error ( 'Erro ao buscar dados:' , err);
  } else {
    console . log ( 'Dados recebidos:' , data);
  }
}

// Usando a função fetchData com um retorno de chamada
fetchData ( 'https://randomUrl.com/data' , handleData);
```

Neste exemplo, a função fetchData busca dados da URL fornecida e chama a função de retorno de chamada com o resultado. handleData é a função de retorno de chamada que processa os dados buscados ou manipula quaisquer erros.

Argumentos para o retorno de chamada :

err: Um objeto de erro se houve um erro ao buscar os dados, caso contrário, nulo. data: Os dados buscados da API se bem-sucedidos.

TEMPORIZADOR COM SETTIMEOUT

```
function  greet ( name, callback ) {
  setTimeout ( () => {
    console . log ( 'Hello,' + name);
    callback ();
  }, 2000 );
}

// Função de retorno de chamada
function  afterGreeting ( ) {
  console . log ( 'Greeting completed.' );
}

// Usando a função greet com um retorno de chamada
greet ( 'Mohi' , afterGreeting);
```

Neste exemplo, a função greet registra uma mensagem de saudação após um atraso de 2 segundos e, em seguida, chama a função de retorno de chamada. afterGreeting é a função de retorno de chamada executada após a saudação ser registrada.


PROBLEMAS COM
CALLBACKS:
CALLBACK HELL

O Callback Hell, também conhecido como Pyramid of Doom, é um termo utilizado para descrever a situação em que o código JavaScript se torna difícil de ler e manter devido ao aninhamento excessivo de callbacks. Em Node.js, a maioria das operações de I/O, como leitura de arquivos ou chamadas de API, são assíncronas e baseadas em callbacks. Isso significa que, em vez de esperar pela conclusão de uma operação, o código continua a ser executado e um callback é chamado quando a operação é concluída.

```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```

O problema surge quando várias operações assíncronas precisam ser executadas em sequência, uma após a outra. Cada operação requer um callback para lidar com o resultado, e isso resulta em um aninhamento excessivo de callbacks. O código se torna difícil de ler, entender e manter, tornando-se um verdadeiro “inferno” para os desenvolvedores.

ESTRATÉGIAS PARA LIDAR COM O CALLBACK HELL

Felizmente, existem várias estratégias que os desenvolvedores podem adotar para lidar com o Callback Hell em Node.js. Uma das abordagens mais comuns é o uso de bibliotecas de promessas, como o Bluebird ou o Q. As promessas são objetos que representam o resultado de uma operação assíncrona e permitem que o código seja escrito de forma mais linear, evitando o aninhamento excessivo de callbacks.
Além disso, o uso de sintaxe async/await, introduzida no ECMAScript 2017, também pode ajudar a lidar com o Callback Hell. Essa sintaxe permite que o código assíncrono seja escrito de forma síncrona, tornando-o mais legível e fácil de entender. No entanto, é importante destacar que o suporte a async/await pode variar entre as versões do Node.js e os navegadores.

```
async (() => {
    let result = await function foo(abc){
        // ....
    };
    //play with result
    let resultOfX = await function x(result){
        //....
    }
    // play with resultOfX
})
```


BORA
COOP

NOSSOS CONTATOS:

27 99500-7495

https://beacons.ai/prismatech

producaoprismatech@gmail.com

Avenida Jerônimo Monteiro 145, Vitória