Základní nástroje pro paralelní a distribuované systémy v jazyce JavaScript

Verze: 28. 11. 2024

TIP: https://dummyjson.com/docs
DummyJSON lze použít u jakéhokoliv typu front end projektu, který potřebuje produkty, košíky, uživatele, todos nebo jakákoli fiktivní data ve formátu JSON.

Asynchronní programování pomocí then/catch/finally

Dokumentace then
Dokumentace catch
Dokumentace finally

```
1  // Zadání požadavku na uvedenou adresu
2  fetch("//dummyjson.com/test")
3    .then(res => res.json()) // Převedení zdroje na JSON
4    .then(console.log) // Zobrazení výsledku v konzoli
5    .catch((err) => alert(err)) // Zachycení chyby
6    .finally(console.log("Done."))
7
8  > {
9      "status": "ok",
10     "method": "GET"
11 }
12 > Done.
```

Vysvětlení:
fetch – rozhraní pro zadávání požadavků HTTP a zpracování odpovědí
then – přidání callbacku pro příslib
catch – zachycení erroru v bloku
finally – blok, který se vykoná vždy, nezáleží na předchozím výsledku vyhodnocení

Asynchronní programování pomocí async/await

Dokumentace async
Dokumentace await

```
const req = async function(url){
  try{
    // Zadání požadavku na uvedenou adresu
    const res = await fetch(url)

    // Logika podle výsledku
    if(!res.ok){
      throw new Error("Custom Error")
    }

    // Zpracování a vypsání
    const data = await res.json()
    console.log(data);
  } catch(err){ // Zachycení a vypsání chyby
    console.log(err);
  }
  console.log("Done.")
}

req("//dummyjson.com/test")

> {
    "status": "ok",
    "method": "GET"
}
> Done.
```

Vysvětlení:
async – funkce, která vrací **Promise**
await – čekání na vyřešení **Promise** (asynchronního kódu)

Vytvoření příslibu

[Dokumentace Promise]

```
// Vyvoření nového příslibu
const promiseA = new Promise((resolve, reject) => {
  resolve(777);
  // reject(new Error('rejected'))
});

promiseA
  .then(val => console.log("value:", val))
  .catch(e => console.log(e))

> "value:" 777
```

Web Workers

- Neumí pracovat s **UI**, poslat **Alert**, změnit **Session**
- Změny **UI** musí zařídit hlavní vlákno
- **CPU** bound úlohy
- Dokumentace

Dokumentace SharedWorkeru
Dokumentace ServiceWorkeru
Dokumentace DedicatedWorkeru

ServiceWorker

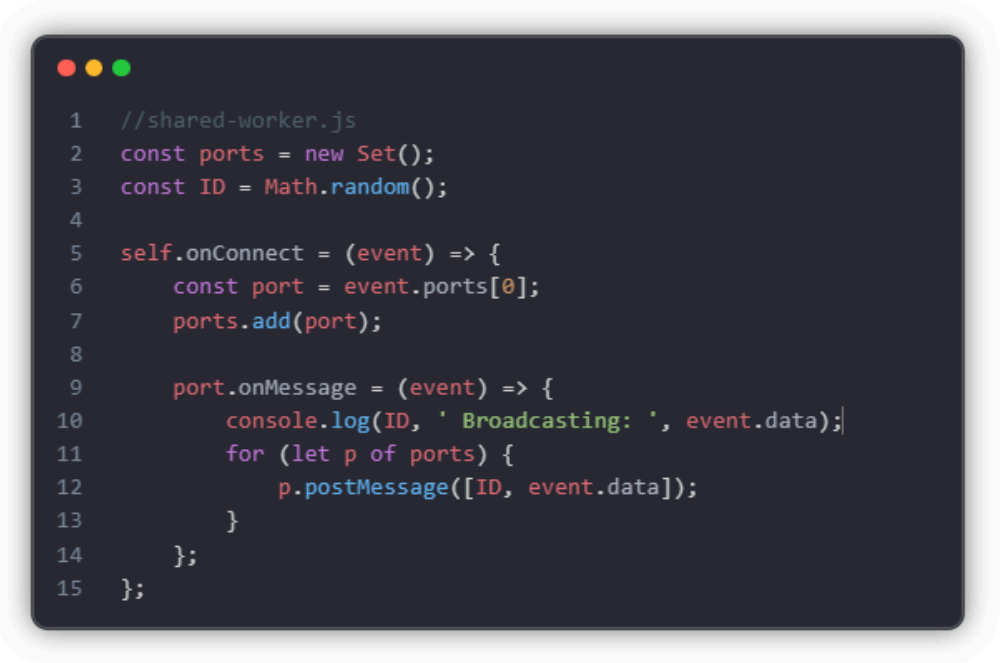
- **Proxy** mezi **webovou aplikací** a **serverem**
- Caching, offline formuláře

```
const CACHE_NAME = 'my-cache1';

const FILES_TO_CACHE = [
  '/',
  '/index.html',
  '/styles.css',
  '/script.js',
  '/offline.html',
];

self.addEventListener('install', (event) => {
    event.waitUntil(
        caches.open(CACHE_NAME).then((cache) => {
            return cache.addAll(FILES_TO_CACHE);
      })
    );
  });
```

```
// Return from cache otherwise fetch it from the network
self.onFetch = (event) => {
    console.log('Fetching:', event.request.url);
    event.respondWith(
        caches.match(event.request).then((response) => {
            return response || fetch(event.request);
        })
    );
};
```

Shared Worker

- Sdílený mezi stránkami, často slouží jako SINGLETON
- konzistence v otevřeném dokumentu na více strankách
- Broadcasting:



```
//shared-worker.js
const ports = new Set();
const ID = Math.random();

self.onConnect = (event) => {
    const port = event.ports[0];
    ports.add(port);

    port.onMessage = (event) => {
        console.log(ID, ' Broadcasting: ', event.data);
        for (let p of ports) {
            p.postMessage([ID, event.data]);
        }
    };
};
```

Dedicated Worker

- Dedikovaný pro konkrétní úlohu, nelze sdílet
- Zpracování zvuku, obrazu, textu bez možnosti BE

```

onmessage = function(event) {
    console.log('Worker recieved message: ' + event.data);
    const result = fibonacci(event.data);
    console.log('It is calculated, sending to main thread!');
    postMessage(result);
};


function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Synchronizace v Node.js

Node.js nabízí několik možností pro paralelní zpracování úloh. Mezi ty hlavní patří použití `child_process`, `worker_threads` a `cluster`.

child_process

Tento modul umožňuje vytvářet nové procesy na úrovni operačního systému. Tyto procesy běží nezávisle na hlavním procesu. Je tedy vhodný pro spouštění externích příkazů nebo pro zpracování, které může běžet mimo hlavní proces. Každý vytvořený proces má svou vlastní paměť a event loop. Nabízí čtyři základní metody:

- **`spawn`**: Umožňuje spustit nový proces a průběžně komunikovat přes standardní vstup/výstup
- **`exec`**: Spouští příkaz v shellu a vrátí výstup jako callback
- **`execFile`**: Spustí soubor přímo (bez shellu)
- **`fork`**: Speciálně navržený pro komunikaci mezi rodičem a potomkem

woker_threads

Modul `worker_threads` umožňuje použití více vláken uvnitř jednoho procesu. Každý `Worker` běží ve vlastním vlákně a má svůj vlastní event loop, nezablokuje tedy hlavní event loop. Od `child_process` se liší tím, že umožňují sdílení paměti mezi vlákny pomocí `SharedArrayBuffer` a `Atomics`.
`SharedArrayBuffer` je buffer, který je dostupný všem vláknům. Abychom zabránili chybě souběhu, tak můžeme použít globální objekt `Atomics`, který poskytuje různé metody pro atomické operace. Tyto operace zajišťují, že změny prováděné na sdílených datech budou vždy konzistentní i když k nim přistupuje více vláken současně. Mezi hlavní metody patří:

- **`Atomics.add`**: Přidá hodnotu k číslu uloženému ve sdíleném bufferu na konkrétním indexu a vrátí původní hodnotu
- **`Atomics.sub`**: Odečte hodnotu od čísla uloženého ve sdíleném bufferu na konkrétním indexu a vrátí původní hodnotu
- **`Atomics.load`**: Vrátí hodnotu ve sdíleném bufferu na konkrétním indexu
- **`Atomics.store`**: Uloží hodnotu ve sdíleném bufferu na konkrétním indexu
- **`Atomics.wait`**: Blokuje vlákno, dokud se hodnota na konkrétním indexu nezmění
- **`Atomics.notify`**: Probudí čekající vlákna na konkrétním indexu

Příklad worker_threads se sdílenou proměnnou:

```js
// main.js
const { Worker } = require('worker_threads');

const buffer = new SharedArrayBuffer(4);
const sharedArray = new Int32Array(buffer);

sharedArray[0] = 0;

function createWorker() {
    return new Promise((resolve) => {
        const worker = new Worker('./worker.js', {
            workerData: { buffer }
        });
        worker.on('message', (message) => {
            console.log(`Worker message: ${message}`);
            resolve();
        });
    });
}

async function main() {
    await Promise.all([createWorker(), createWorker()]);
    console.log(`Final counter value: ${sharedArray[0]}`);
}

main();
```

```js
// worker.js
const { workerData, parentPort } = require('worker_threads');

const { buffer } = workerData;
const sharedArray = new Int32Array(buffer);

for (let i = 0; i < 1_000_000; i++) {
    Atomics.add(sharedArray, 0, 1);
}

if (parentPort) {
    parentPort.postMessage('Worker finished');
}

```

cluster

Modul `cluster` umožňuje vytvářet více instancí Node.js serveru. Cluster vytvoří `master` proces a ten spravuje více `worker` procesů. Každý worker může samostatně obsluhovat požadavky nezávisle na ostatních workerech, to umožňuje distribuovat zátěž a škálovat aplikaci.
Příklad využití cluster modulu:

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
    for (let i = 0; i < numCPUs; i++) {
        cluster.fork();
    }

    cluster.on('exit', (worker, code, signal) => {
        console.log(`Worker ${worker.process.pid} ukončen`);
    });
} else {
    http.createServer((req, res) => {
        res.writeHead(200);
        res.end('Hello, world!\n');
    }).listen(8000);

    console.log(`Worker ${process.pid} spuštěn`);
}
```

Knihovny pro paralelní zpracování

JavaScript je jednovláknový jazyk, ale knihovny jako **Parallel.js**, **Thread.js**, a **RxJS** umožňují efektivní zpracování úloh pomocí paralelismu a asynchronního programování.

Parallel.js

[Dokumentace Parallel.js]

Parallel.js umožňuje snadné paralelní zpracování dat ve webovém prostředí i Node.js pomocí mapování, redukce a dalších metod. Používá Web Workers k paralelnímu vykonávání úloh.

new Parallel()

Vytvoří nový paralelní proces, který může přijímat funkci, kterou bude vykonávat na více vláknech.

Příklad:

```
const p = new Parallel([1, 2, 3, 4]);
p.map(x => x * 2).then(result => console.log(result)); // [2, 4, 6, 8]
```

map()

Provádí operaci na každém prvku pole paralelně.

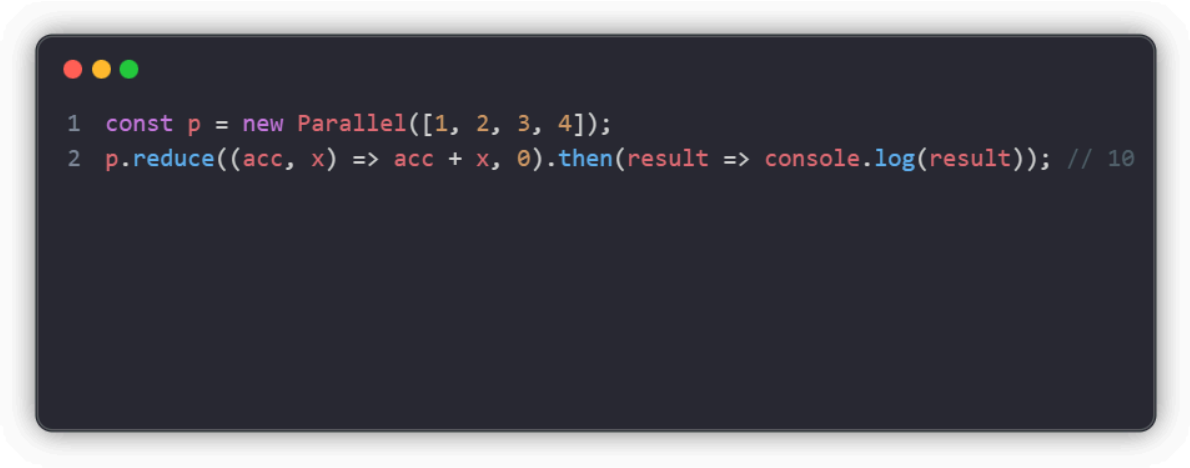
Příklad:


```
const p = new Parallel([1, 2, 3, 4]);
p.map(x => x * 2).then(result => console.log(result)); // [2, 4, 6, 8]
```


reduce()

Provádí redukci pole (sčítání, součty, atd.) na základě funkce.

Příklad:



filter()

Filtruje hodnoty v poli na základě podmínky.

Příklad:

Thread.js

[Dokumentace Thread.js]

Thread.js usnadňuje vytváření a správu vláken (workers) pro paralelní výpočty v JavaScriptu. Podporuje moderní syntax a nabízí snadnou komunikaci mezi vlákny.

spawn()

Vytváří nové vlákno (worker) a vrací pracovní instanci pro komunikaci.

Příklad:

```
const { spawn, Thread } = require('threads');
const worker = await spawn(new Worker('./worker'));
const result = await worker.someFunction();
await Thread.terminate(worker);

```

expose()

Exponuje metody, které budou dostupné pro volání z hlavního vlákna.

Příklad:

```
const { expose } = require('threads/worker');
expose({
  someFunction: (input) => {
    return input * 2;
  }
});

```

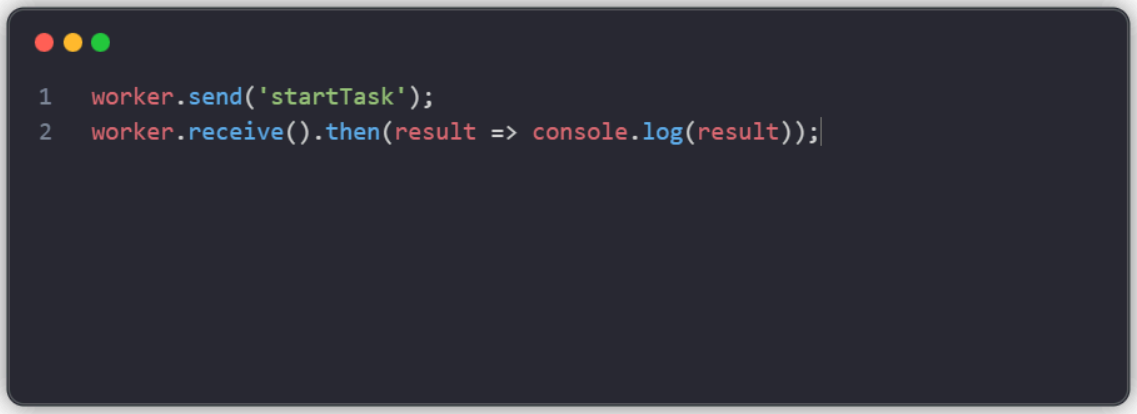
Thread.terminate()

Ukončí vlákno (worker) a uvolní všechny zdroje.

send() a receive()

Slouží pro posílání a příjem zpráv mezi hlavním vláknem a pracovními vlákny.

Příklad:



```
worker.send('startTask');
worker.receive().then(result => console.log(result));
```

RxJS

[Dokumentace RxJS]

RxJS je knihovna pro reaktivní programování, která umožňuje zpracovávat asynchronní datové proudy. Nabízí širokou škálu operátorů pro transformaci a manipulaci s proudy a je často využívána v aplikacích s dynamickými daty a událostmi.

of()

Vytváří nový Observable z předaných hodnot.
Příklad:

```
import { of } from 'rxjs';
const observable = of(1, 2, 3);
observable.subscribe(value => console.log(value)); // 1, 2, 3
```

from()

Vytváří Observable z iterovatelného objektu (například pole, Promise, nebo NodeList).

Příklad:

```
import { from } from 'rxjs';
const observable = from([10, 20, 30]);
observable.subscribe(value => console.log(value)); // 10, 20, 30
```

map()

Transformuje hodnoty emitované Observable.
Příklad:

```
import { map } from 'rxjs/operators';
const observable = of(1, 2, 3);
observable.pipe(map(x => x * 2)).subscribe(value => console.log(value)); // 2, 4, 6

```

filter()

Filtruje hodnoty podle zadané podmínky.

Příklad:

```
import { filter } from 'rxjs/operators';
const observable = of(1, 2, 3, 4, 5);
observable.pipe(filter(x => x % 2 === 0)).subscribe(value => console.log(value)); // 2, 4
```

subscribe()

Používá se k odběru (subscription) hodnot z Observable a spouští příslušnou akci, jakmile jsou hodnoty emitovány.
Příklad:

```
observable.subscribe(value => console.log(value)); // výstup: 2, 4, 6
```

mergeMap() a switchMap()

Používají se k mapování hodnot do nových Observablů a řízení vnořených asynchronních operací.

Příklad:

```
import { switchMap } from 'rxjs/operators';
observable.pipe(
  switchMap(value => apiCall(value))
  // Například API volání na základě hodnoty).subscribe(result => console.log(result));
```