JavaScript

Moduli - NPM- Multithreading/Multiprocessing

Giovanni Iovino: giovanni.iovino@intecs.it

Contenuto:

Esecuzione di codice JS in thread/processi separati, moduli js.

Modules

what are modules?

- In JavaScript un modulo è un file. Ogni script JS è un modulo!
- Un modulo può caricare altri moduli per usarne le funzionalità
- I moduli funzionano in strict-mode by default
- Ogni modulo ha uno **scope dedicato**, le variabili e le funzioni dichiarate in un modulo sono visibili solo all'interno del modulo e non negli altri moduli.
- Per rendere visibili le variabili/funzioni di un modulo ad altri moduli si usano direttive speciali di import/export
- All'interno di un modulo la variabile **this** non è definita!
- Il codice del modulo viene eseguito solo nel momento in cui il modulo è importato

Modules

what are modules?

- Oggi Javascript ha un sistema nativo di gestione dei moduli (E2015 Modules)
- Tuttavia esistono altri standard di gestione dei moduli in Javascript tuttora utilizzati:
  - **CommonJs** (Node.js di default usa i moduli nel formato CommonJS)
  - AMD (Asynchronous Module Definition)
  - UMD (Universal Module Definition)

ES2015 Modules (ESM)

- Il sistema nativo di gestione dei moduli ESM utilizza le direttive **import** e **export** rispettivamente per l'importazione dei moduli e la definizione dell'interfaccia pubblica del modulo
- Non avviene il wrapping del modulo in una funzione (questa differenza è uno dei motivi per cui se ne è ritardata l'adozione)
- Nodejs utilizza di default CommonJs, ma supporta ESM. Per poter utilizzare ESM i moduli devono avere estensione **.mjs** invece di **.js**

https://nodejs.org/api/esm.html

```
// add.mjs
function add(a,b){
    return a+b;
}
function sub(a,b){
    return a-b;
}
export function mul(a,b){
  return a*b;
}
export const PI = 3.14;

const TEN = 10;

export default TEN;

export {add, sub}
```

```
// index.mjs
import {add,sub, mul, PI} from './esm_add.mjs'
import TENNNN from './esm_add.mjs'

const s = add(TENNNN,PI);
console.log(s);
```

CommonJs

- CommonJs incapsula il codice di un modulo all'interno di una funzione(1) esponendo l'interfaccia pubblica del modulo **module.exports**
- Per importare un modulo in un altro modulo si usa la direttiva **require**(2)

https://nodejs.org/docs/latest/api/modules.html

```
// add.js
function add (a, b) {
  return a + b
}
module.exports = add
```

(1)

```
(function (exports, require, module, __filename, __dirname) {
  function add (a, b) {
    return a + b
  }

  module.exports = add
})
```

(2)

```
// index.js
const add = require('./add')

console.log(add(4, 5))
```

NPM

Node Package Manager

- NPM è il package manager predefinito di Node.js, installato automaticamente quando si installa node.
- NPM è principalmente:
  - Un registro pubblico di pacchetti software utilizzabili in Node.js www.npmjs.com
  - Un CLI (Command Line Interface) con cui gestire i nostri progetti e le loro dipendenze

https://docs.npmjs.com/about-npm

NPM CLI

Node Package Manager CLI

- Il CLI npm ci permette di:
  - Creare un progetto: **npm init**
  - Aggiungere moduli npm: **npm install**
  - Rimuovere moduli npm: **npm uninstall**
  - Eseguire script: **npm run ... npm start**
  - Pubblicare un pacchetto sul registry npm
  - ... e molto altro

Per una una guida esaustiva fare riferimento a

https://docs.npmjs.com/cli/v7

NPM Package

package.json

- Tutti i packages npm sono descritti in un file chiamato **package.json**
- Il file contiene i metadati necessari alla configurazione del nostro progetto e alla gestione delle dipendenze

Example package.json

```
{
  "name" : "underscore",
  "description" : "JavaScript's functional programming helper library.",
  "homepage" : "http://documentcloud.github.com/underscore/",
  "keywords" : ["util", "functional", "server", "client", "browser"],
  "author" : "Jeremy Ashkenas <jeremy@documentcloud.org>",
  "contributors" : [],
  "dependencies" : [],
  "repository" : {"type": "git", "url": "git://github.com/documentcloud/underscore.git"},
  "main" : "underscore.js",
  "version" : "1.1.6"
}
```

Single threaded

at least the Main Thread

- Ogni applicazione js viene eseguita in un processo separato (es. nei browser uno per ogni pagina)
- Il codice che scriviamo viene eseguito in un singolo thread (Main thread)

"Significa che tutto il codice eseguito è single-threaded?
Non necessariamente. Ad esempio alcuni moduli nativi di Node.js possono essere eseguiti automaticamente in thread separati ..."

(per saperne di più https://github.com/libuv/libuv)

JS

Single threaded

at least the Main Thread

- La natura non bloccante del main thread in Javascript risulta particolarmente efficiente per le applicazioni con operazioni intensive di I/O. (Es. server che gestiscono migliaia di client)
- Le operazioni che richiedono un uso intensivo di CPU, al contrario, rischiano di bloccare l'esecuzione dell'applicazione (modello 'run-to-completion' dell'esecuzione delle funzioni)

CPU blocking example

- L'esecuzione di cpuFun blocca l'event-loop, impedendo a setInterval di essere eseguita ad intervalli regolari dal momento in cui viene eseguita finché non termina

```
let n = 0;

//Funzione che richiede qualche secondo per essere completata
const cpuFun = function (){
    const start = new Date().getTime();
    let a = 0;
    for(let i=0; i<10000000000; i++){
      a = a*i;
    }
    return (new Date().getTime()-start);

};

//Eseguiamo una funzione ad intervalli di 100 ms per 3 secondi circa
let start = new Date().getTime();
const tid = setInterval(()=>{
    console.log(n);
    n++;
    if(n===30){
        clearInterval(tid);
        console.log("setInterval cleared after",
        (new Date().getTime()-start),"ms");
    }
},100);

//Eseguiamo la funzione CPU-intensive dopo 500 ms
setTimeout(()=>{
    const longRes= cpuFun();
    console.log("long cpu fun required ", longRes, "ms to run");
}, 500);
```

Child_process

Creazione di sotto-processi (node.js)

- Modulo disponibile in node.js
- Permette la creazione di processi separati in cui eseguire codice JS o programmi esterni tramite i metodi:
  - Spawn
  - Exec
  - Fork
- Ogni processo ha il suo spazio di memoria dedicato, e nel caso, un'istanza dell'engine Javascript dedicata
- La comunicazione tra processo padre e processo figlio avviene tramite IPC (Inter Process Communication) ed eventi. NON c'è memoria condivisa tra processi diversi

Reference:
https://nodejs.org/api/child_process.html

JS

Child process - Spawn

Sintassi: **child_process.spawn(command[, args][, options])**

- Esegue un comando esterno in un processo separato.
- Vengono creati dei pipe automatici con gli stream stdout, stdin e stderr
- E' possibile usare il paradigma basato sugli eventi per interagire con il processo esterno

```
const { spawn } = require('child_process');


const childProcess =
spawn("C:\\Windows\\System32\\PING.exe",
["www.google.com"]);

childProcess.stdout.on("data", message => {
    console.log("childprocess output:");
    console.log(message);
});

childProcess.on("close", message => {
    console.log("childprocess closed with code",
message);
});

childProcess.on("error", message => {
    console.log("childprocess error", message);
});
```

Child process - Exec

Sintassi: **child_process.exec(command[, options][, callback])**

- Esegue un comando esterno all'interno di una shell di sistema.
- Vengono creati dei pipe automatici con gli stream stdout, stdin e stderr
- E' possibile usare il paradigma basato sugli eventi per interagire con il processo esterno

```
const { exec } = require('child_process');

const childProcess = exec("dir");
childProcess.stdout.on("data", message => {
    console.log("childprocess output:");
    console.log(message);
});
childProcess.on("close", message => {
    console.log("childprocess closed with
code", message);
});

childProcess.on("error", message => {
    console.log("childprocess error", message);
});
```

JS

Child process - Fork

Sintassi:**child_process.fork(modulePath[, args][, options])**

- Esegue lo spawn di un nuovo processo node.js per l'esecuzione di un modulo Javascript
- La comunicazione tra processo padre e processo figlio avviene tramite IPC
- Il processo figlio utilizza la variabile globale **process** (disponibile in node.js) per agganciare gli eventi e le chiamate IPC (https://nodejs.org/api/process.html)

Child process - Fork

child, modulo cpu_f.js:

```
process.on("message", message => {
    const result = cpuFun()
    process.send(result)
    process.exit() // make sure to use exit()
})

const cpuFun = ()=>{
    const start = new Date().getTime();
    let a = 0;
    for(let i=0; i<10000000000; i++){
        a = a*i;
    }
    return (new Date().getTime()-start);
}
```

parent:

```
const { fork } = require('child_process');

let n = 0;
let start = new Date().getTime();
const tid = setInterval(() => {
    console.log(n);
    n++;
    if (n === 30) {
        clearInterval(tid);
        console.log("setInterval cleared after", (new
Date().getTime()-start),"ms");
    }
}, 100);

const childProcess = fork("./fork/cpu_f.js");
childProcess.send(null) //send method is used to send
message to child process through IPC
childProcess.on("message", message => { //Event
triggered when a message is received from the child
process
    console.log("long cpu fun required ", message, "ms
to run");
});
```

Cluster

Creazione di sotto-processi (node.js)

- Modulo disponibile in node.js
- Permette la creazione di processi (chiamati worker) in cui eseguire codice JS.
- Internamente utilizza child_process.spawn per la creazione dei sottoprocessi.
- I workers possono condividere la stessa porta TCP (Usato normalmente come primo approccio per scalare i server HTTP in node - loadbalancing)
- https://nodejs.org/api/cluster.html

Reference: https://nodejs.org/api/cluster.html

(Web)Workers

Creazione di thread (node.js e browser)

- Modulo disponibile in node.js e nei browser
- Permette la creazione di thread separati all'interno del processo principale in cui è eseguito il codice JS.

Reference:

https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

https://nodejs.org/api/worker_threads.html

(Web)Workers

Creazione di thread (node.js e browser)

Invece di avere:

- Un singolo processo
- Un singolo thread
- Un event-loop
- Una istanza dell'engine JS
- (Una singola istanza di Node.js)

Utilizzando i worker threads si ha:

- Un singolo processo
- Threads multipli
- Un event loop per thread
- Una istanza di engine JS per thread
- (Una istanza di Node.js per thread)

Worker - NodeJs

child, modulo cpu_f.js:

```
const { parentPort} = require('worker_threads');

parentPort.on("message", message => {
    const result = cpuFun();
    parentPort.postMessage(result);
    process.exit() // OK! in un thread termina il thread
})

const cpuFun = ()=>{
    const start = new Date().getTime();
    let a = 0;
    for(let i=0; i<10000000000; i++){
        a = a*i;
    }
    return (new Date().getTime()-start);
}
```

parent:

```
const { Worker} = require('worker_threads');

let n = 0;
let start = new Date().getTime();
const tid = setInterval(() => {
    console.log(n);
    n++;
    if (n === 30) {
        clearInterval(tid);
        console.log("setInterval cleared after", (new
Date().getTime()-start),"ms");
    }
}, 100);

const w = new Worker("./cpu_f.js");
w.postMessage(null) //send method is used to send message to child
process through IPC
w.on("message", message => { //Event triggered when a message is
received from the child process
    console.log("long cpu fun required ", message, "ms to run");
});
```