Concurrència en nodejs

IES Jaume II El Just
Tavernes de la Valldigna

Aplicacions multiplataforma amb tecnologies web

Curs 20-21

Continguts

1. Concurrència en nodejs

Javascript és un llenguatge **asíncron i concurrent**. Què volíen dir estos dos conceptes?

- **Asíncron**: La programació tradicional es basa en el paradigma de *programació sequencial*, on un programa és una seqüència d'instruccions que s'executen ordenadament, i una operació no comença fins que no acaba l'anterior. Es tracta doncs d'un model síncron. Front a açò, la *programació asíncrona* dóna la possibilitat que algunes operacions tornen el control de l'execució al programa principal abans d'haver acabat, la qual cosa agilitza el procés d'execució.
- **Concurrent**: La *concurrència* es donava quan només una tasca s'estava executant en un moment donat, front al paral·lelisme, on hi havia diverses tasques executant-se al mateix temps.

Aleshores, el fet que javascript siga asíncron i concurrent significa que les instruccions no tenen per què seguir un ordre seqüencial, de manera que no cal esperar que acaben certes instruccions per iniciar l'execució d'altres, i que a més, només s'està executant una instrucció a la vegada.

Per a tot açò, Javascript (i per tant nodejs) fan ús del que es coneix com *event loop* o el bucle d'events, que és el component encarregat de coordinar l'execució, els events i els callbacks. Veiem alguns d'aquests conceptes amb un exemple:

```
1  function saluda(nom){
2  console.log("Hola "+nom);
3  }
4
5  for (let i=0;i<5;i++){
6      setTimeout(function (){
7          saluda(i);
8      }, 500-i*100);
9  }
10
11 process.nextTick(function(){saluda("Prioritari 1")});
12 process.nextTick(function(){saluda("Prioritari 2")});
13
14 console.log("Fi de l'execució");
```

Analitzem l'exemple:

- Definim una funció *saluda*, que rep un paràmetre *nom*, i escriu a la consola «Hola», seguit del nom que li passem.
- Posteriorment, llancem un bucle que compta de 0 a 5, i dins, defineix un timeOut. La funció setTimeOut el que fa és posar en marxa un temporitzador, que llança una funció en vèncer aquest:

```
1   setTimeout(function, temps_en_ms);
```

- Com veiem, la funció que s'executa en vèncer el temporitzador (es tracta d'una funció anònima, ja que no te nom), el que fa és invocar a la funció *saluda*, passant-li com a paràmetre el valor *i* de control del bucle. Aquestes funcions que es llancen en resposta a un event (com és el cas de vèncer el temporitzador) s'anomenen funcions de retorn o de *callback*.
- A més, al segon paràmetre de la funció setTimeOut li passem el temps que tardarà en vèncer aquest. En l'exemple, hem indicat que aquest temps siga major com més menut siga el valor de i, segons la fòrmula 500-i*100.
- S'han afegit al final dues instruccions `process.nextTick`, que afigen dues funcions també a executar. Després aprofundirem més en aquestes funcions.
- I finalment, mostrem per pantalla el missatge que ha finalitzat l'execució del programa principal.

Si llancem ara l'script, el resultat és el següent:

```
$ nodejs exemple1.js
Fi de l'execució
hola Prioritari 1
hola Prioritari 2
hola 4
hola 3
hola 2
hola 1
hola 0
```

Si ens adonem, fa la sensació que les ordres s'han executat en ordre pràcticament invers al que esperaríem en la programació sequencial tradicional. Anem a veure el per què de tot açò.

1.1. L'event Loop o bucle d'esdeveniments

Nodejs gestiona la concurrència mitjançant un únic fil d'execució, de manera que el primer que s'executa és el programa principal. Si durant l'execució d'aquest programa principal es produeixen certs esdeveniments, aquests s'afigena una cúa d'esdeveniments.

En aquest moment, entra en acció l'*Event Loop*, que és un bucle que està sempre en execució, i que té la missió de vigilar la cúa d'events i llançar els *callbacks* corresponents. Quan un callback acaba, torna el control al bucle d'esdeveniments, qui ja s'encarregarà de gestionar els següents events. Si durant l'execució d'aquesta es produeix altre esdeveniment, aquest s'afig a la cúa, però no interromp l'execució de l'actual; d'aquesta manea, ens assegurem que una funció en resposta a un esdeveniment comença i acaba, evitant-nos així accessos concurrents a seccions crítiques i altres problemes de concurrència de la programació seqüencial. Quan la cúa d'esdeveniments és buida, es torna el control al sistema operatiu.

Podem trobar més informació als següents vídeos:

- Gestión de la concurrencia, bucle de eventos y nextTick del MOOC «Desarrollo de Aplicaciones con HTML5, node.js y JavaScript».
- Asincronía en JavaScript, dde appdelante.com (quatre parts).

I als enllaços:

- Entendiendo la magia detrás de NodeJs y su Event Loop
- Historia del Callback Hell en Node.js

A més, al lloc web de Philipp Roberts, podem trobar l'eina loupe, que ens permet observar el comportament de la cúa d'esdeveniments de JS.

2. L'objecte Process de nodejs

Process és un objecte (de la classe EventEmitter) global que ens ofereix informació i control sobre l'actual procés en Node.js. Com que es tracta d'un objecte global, està disponible en totes les aplicacions Node.js, sense necessitat d'importar-lo mitjançant `require` (l'equivalent a l'**`import`** de java)

A l'API de nodejs podem trobar tota la informació sobre aquest objecte. En aquest apartat veurem alguns dels mètodes més destacats.

2.1. Atributs

Atribut	Descripció
process.arch	Retorna l'arquitectura del sistema operatiu
proess.argv	Vector amb els arguments de la línia d'ordres
process.env	Conté un objecte amb les variables d'entorn de l'usuari (env)
process.pid	Conté el PID del procés actual
process.ppid	Conté el PID del procés pare
process.platform	Conté el sistema operatiu que s'està executant
process.stdin	Conté un objecte de tipus stream connectat a l'entrada estàndard
process.stdout	Conté un objecte de tipus stream connectat a l'eixida estàndard

Atribut	Descripció
process.sterr	Conté un objecte de tipus stream connectat a l'eixida d'error estàndard
process.title	Conté el nom del procés actual

2.2. Mètodes

Mètode	Descripció
process.cpuUsage()	Torna la memòria d'usuari i del sistema utilitzada pel procés
process.memoryUsage()	Retorna la quantitat de memòria utilitzada pel procés de Node.js
process.cwd()	Retorna el directori de treball del procés
process.chdir(dir)	Canvia el directori de treball del procés
process.edit([codi])	Ix del procés de forma síncrona, amb determinat estat d'eixida
process.kill(pid, [senyal])	Envía una senyal a un procés identificat pel pid
process.uptime	Retorna el número de segons que es troba actiu l'actual procés de nodejs

3. El mòdul OS

El mòdul OS ens ofereix algunes funcions relacionades amb el sistema operatiu. Podem trobar l'API completa en: https://nodejs.org/api/os.html.

Alguns dels mètodes i atributs més interessants són:

Atribut	Descripció
os.EOL	Obté el caràcter de final de línia del sistema operatiu

Mètode	Descripció
os.cpus()	Obté un vector amb informació sobre les diferents CPUs
os.homedir()	Retorna el home de l'usuari actual
os.hostname()	Retorna el hostname

4. El mòdul Child Process

El mòdul Child Process, ofereix la possibilitat de crear processos fills i executar ordres del sistema, de forma semblant a com feia el ProcessBuilder de Java.

Veiem amb un exemple com executaríem una ordre del sistema (extret de la documentació oficial de nodejs):

```
1
2  const { spawn } = require('child_process');
3  const ls = spawn('ls', ['-lh', '/usr']);
4
5  ls.stdout.on('data', (data) => {
6    console.log(`stdout: ${data}`);
7  });
8
9  ls.stderr.on('data', (data) => {
10   console.log(`stderr: ${data}`);
11 });
12
13 ls.on('close', (code) => {
14   console.log(`child process exited with code ${code}`);
15 });
```