Seminari 1 – Node.js

Guia d'estudi (Part 2). Callbacks i Promeses

1. Introducció

Aquesta guia estén les seccions de la presentació sobre JavaScript/Node.js relatives a les funcions asincròniques amb callbacks i amb promeses. La següent secció, titulada *Callbacks*, s'organitza en dues parts: en la primera, s'explica la implementació de les funcions callback per al seu ús en les invocacions de funcions asincròniques; en la segona, s'explica com el programador pot desenvolupar funcions que tinguen comportament asincrònic en Node, mitjançant l'estudi d'alguns problemes concrets. En l'última secció d'aquest document, titulada *Promeses*, s'explica aquesta alternativa de la programació asincrònica, abordant els mateixos problemes resolts anteriorment amb callbacks.

2. Callbacks

2.1. Implementació de callbacks per a funcions asincròniques

La programació asincrònica en Node.js es basa en l'ús de les funcions callback. Quan un programa conté la invocació d'una funció asincrònica, per exemple ***f_async***, que porte un callback com a últim argument, aquest programa no es bloqueja fins que ***f_async*** concloga, sinó que continuarà executant les instruccions següents que tinga. Mentrestant, la funció asincrònica s'anirà executant i, a la seua fi, en un torn posterior, s'executarà el seu callback. Sobre aquest tema, vegeu la fulla 37 de la presentació del seminari.

En "Control Flow in Node"[1], Tim Caswell presenta alguns exemples interessants per a comprendre el funcionament de les funcions asincròniques i els callbacks. El següent codi, amb lleus modificacions, està pres d'un exemple de Caswell.

```
 1: var fs = require('fs');
 2:
 3: fs.readFile('mydata.txt', function (err, buffer) {
 4:   if (err) {
 5:     // Handle error
 6:     console.error(err.stack);
 7:     return;
 8:   }
 9:   // Do something
10:   console.log( buffer.toString() );
11: });
12:
13: console.log('executant altres instruccions');
14: console.log('arrel(2) =', Math.sqrt(2));
```

La crida a la funció asincrònica ***readFile*** del mòdul ***fs*** no bloqueja aquest programa. Les instruccions de les línies 13 i 14 s'executaran abans que es complete ***readFile*** i, per tant, abans de la funció anònima de callback que rep. Si l'intent de lectura del fitxer 'mydata.txt' fallara, en el callback s'executa la instrucció de la línia 6. En tal cas, l'eixida d'aquest programa seria:

[1] Control Flow in Node - Tim Caswell: http://howtonode.org/control-flow, http://howtonode.org/control-flow-part-ii

```
executant altres instruccions
arrel(2) = 1.4142 ...
Error: ENOENT, open '… /mydata.txt'
```

La funció ***readFIle*** rep un nom de fitxer i "retorna" el seu contingut. En sentit estricte, no retorna res, sinó que ho passa com a argument a la seua funció callback. El segon argument del callback (*buffer* en l'exemple anterior) rep el contingut del fitxer quan l'operació de lectura es duu a terme correctament. El primer argument del callback (*err* en l'exemple) rep el missatge d'error generat quan la lectura falla per alguna causa (com ha succeït en l'anterior exemple d'execució).

Definir la funció de callback com a anònima és usual si només s'utilitza una vegada, com en l'exemple anterior. Però, en cas que s'use més vegades, el recomanable és definir-la amb nom. El següent exemple de Tim Caswell, modificació de l'anterior, il·lustra aquest cas.

```
 1: var fs = require('fs');
 2:
 3: function callback(err, buffer) {
 4:   if (err) {
 5:     // Handle error
 6:     console.error(err.stack);
 7:     return;
 8:   }
 9:   // Do something
10:   console.log( buffer.toString() );
11: }
12:
13: fs.readFile('mydata.txt', callback);
14: fs.readFile('rolodex.txt', callback);
```

La implementació de funcions asincròniques mitjançant callbacks permet la implantació sense un altre límit que les necessitats del programador. En "Accessing the File System in Node.js"[2], Colin Ihrig presenta un exemple d'implantació bastant profunda de callbacks:

```
 1: var fs = require("fs");
 2: var fileName = "foo.txt";
 3:
 4: fs.exists(fileName, function(exists) {
 5:   if (exists) {
 6:     fs.stat(fileName, function(error, stats) {
 7:       fs.open(fileName, "r", function(error, fd) {
 8:         var buffer = new Buffer(stats.size);
 9:
10:         fs.read(fd, buffer, 0, buffer.length, null, function(error, bytesRead, buffer) {
11:           var data = buffer.toString("utf8", 0, buffer.length);
```

[2] Accessing the File System in Node.js - Colin Ihrig: http://www.sitepoint.com/accessing-the-file-system-in-node-js/

```
12:
13:         console.log(data);
14:         fs.close(fd);
15:       });
16:     });
17:   });
18:  }
19: });
```

En aquest exemple, es llig el contingut d'un fitxer amb l'ajuda d'un buffer. El callback de la funció asincrònica ***exists*** del mòdul ***fs*** conté una crida a una altra funció asincrònica, ***stats***, que el seu callback, al seu torn, invoca a una altra funció asincrònica, ***open***, que el seu callback invoca a la funció asincrònica ***read***, la qual també té un callback.

Certament hi ha altres formes de llegir un fitxer: usar ***readFile***, com en els exemples proposats per Caswell, o usar versions sincròniques de les funcions del mòdul ***fs***. La fi de presentar aquest últim programa és mostrar les possibilitats d'implantació de callbacks... i els seus inconvenients. Un alt nivell d'implantació, com s'aprecia, dificulta la lectura del codi. En l'exemple, a més, no es duu a terme cap gestió d'errors. Si això es fera, el codi encara seria més difícil d'interpretar, i de seguir la seua execució. En aquestes situacions, es fa molt convenient l'ús de les promeses com a alternativa en la programació asincrònica.

El niat de callbacks és una manera d'assegurar que les successives operacions (des de la més externa a la més interna) s'executen en seqüència. Però també poden plantejar-se situacions en les quals convinga agrupar un conjunt d'operacions paral·leles. Un exemple el proporciona Tim Caswell (en les pàgines web abans citades): la lectura de tots els fitxers existents en un directori. El codi, amb lleus modificacions, és el següent:

```
 1: var fs = require('fs');
 2:
 3: fs.readdir('.', function (err, files) {
 4:   var count = files.length,
 5:   results = {};
 6:   files.forEach(function (filename) {
 7:     fs.readFile(filename, function (data) {
 8:       console.log(filename, 'has been read');
 9:       results[filename] = data;
10:       count--;
11:       if (count <= 0) {
12:         // Do something once we know all the files are read.
13:         console.log('\nTOTAL:', files.length, 'files have been read');
14:       }
15:     });
16:   });
17: });
```

El callback de la funció asincrònica ***readdir*** rep la llista dels fitxers existents en el directori actual. Amb aquesta llista, i mitjançant l'operador ***forEach***, s'inicia la lectura de cadascun dels

fitxers, mitjançant la funció asincrònica ***readFile***. Així, totes les operacions de lectura s'executen en paral·lel i poden acabar en qualsevol ordre. Conforme conclouen, s'executen els seus respectius callbacks. En aquests, s'actualitza una variable comptador, ***count***, la qual cosa permet detectar quan totes les lectures han acabat per si es vol fer alguna operació posterior.

2.2. Implementació de funcions asincròniques

En els exemples de la secció 2.1 anterior, s'han invocat funcions asincròniques ja existents, funcions del mòdul ***fs***. Altres mòduls estàndard de Node.js proporcionen altres funcions asincròniques. En aquests casos, el programador només necessita invocar-les, proporcionant els arguments adequats (el que implica implementar la funció callback que serà el seu últim argument). En aquesta secció 2.2 es presenta com implementar una funció qualsevol, inicialment sincrònica, perquè tinga un comportament asincrònic.

Considere's el següent codi:

```
 1: // ***  fibo1.js
 2:
 3: function fibo(n) {
 4:    return (n<2) ? 1 : fibo(n-2) + fibo(n-1)
 5: }
 6:
 7: function fact(n) {
 8:    return (n<2) ? 1 : n * fact(n-1)
 9: }
10:
11: console.log('Iniciant execució...')
12: console.log('fibo(40) =', fibo(40))
13: console.log('llançat càlcul fibonacci...')
14: console.log('fact(10) =', fact(10))
15: console.log('llançat càlcul factorial...')
```

Les dues funcions ací definides (la funció ***fibo*** per a calcular el terme enèsim de la successió de Fibonacci i la funció ***fact*** per a calcular el factorial del nombre rebut com a argument) són sincròniques. Per això, les seues invocacions, en les línies 12 i 14 d'aquest codi, bloquegen el programa. L'eixida obtinguda és:

```
Iniciant execució...
fibo(40) = 165580141
llançat càlcul fibonacci...
fact(10) = 3628800
llançat càlcul factorial...
```

A més, la segona línia d'aquesta eixida tarda alguns segons a mostrar-se, donat el cost computacional de la funció ***fibo***. Seria desitjable disposar de versions asincròniques de les funcions, de manera que s'invocaren sense bloquejar el programa en les línies 12 i 14, i els missatges de

les línies 13 i 15 es mostraren immediatament (i s'executaren altres instruccions posteriors, si n'hi haguera alguna).

Una forma senzilla d'implantar aquestes versions asincròniques de les funcions consisteix a afegir ***console.log*** com a funció callback i usar la funció ***setTimeout*** per a organitzar la invocació de les funcions (i passar-les així a la cua d'esdeveniments amb uns temporitzadors adequats). Amb aquestes modificacions, es tindria el següent programa:

```
 1: // ***  fibo2.js
 2:
 3: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 4:
 5: function fibo_back1(n,cb) {
 6:    var m = fibo(n)
 7:    cb('fibonacci('+n+') = '+m)
 8: }
 9:
10: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
11:
12: function fact_back1(n,cb) {
13:    var m = fact(n)
14:    cb('factorial('+n+') = '+m)
15: }
16:
17: console.log('Iniciant execució...')
18: setTimeout( function(){
19:    fibo_back1(40, console.log)
20: }, 2000 )
21: console.log('llançat càlcul fibonacci...')
22: setTimeout( function(){
23:    fact_back1(10, console.log)
24: }, 1000 )
25: console.log('llançat càlcul factorial...')
```

L'eixida obtinguda en executar aquesta segona versió del programa és:

```
Iniciant execució...
llançat càlcul fibonacci...
llançat càlcul factorial...
factorial(10) = 3628800
fibonacci(40) = 165580141
```

A més, només l'aparició de l'última línia d'aquesta eixida presenta un retard apreciable.

Encara que el codi anterior és correcte i funciona com s'esperava, la funció usada com callback, ***console.log***, no s'ajusta al conveni habitual. En Node.js, es recomana que les funciones callback reben, com a primer argument, la informació d'error (si la funció asincrònica fallara) i, com a segon argument, el resultat proporcionat (quan la funció asincrònica acaba correctament).

Si es modifica l'anterior programa tenint en compte aquest conveni, es tindria el següent:

```
 1: // ***  fibo3.js
 2:
 3: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 4:
 5: function fibo_back2(n,cb) {
 6:    var err = eval_err(n,'fibonacci')
 7:    var m   = err ? '': fibo(n)
 8:    cb(err,'fibonacci('+n+') = '+m)
 9: }
10:
11: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
12:
13: function fact_back2(n,cb) {
14:    var err = eval_err(n,'factorial')
15:    var m   = err ? '': fact(n)
16:    cb(err,'factorial('+n+') = '+m)
17: }
18:
19: function show_back(err,res) {
20:    if (err) console.log(err)
21:    else     console.log(res)
22: }
23:
24: function eval_err(n,s) {
25:    return (typeof n != 'number') ?
26:              s+'('+n+') ??? : '+n+' is not a number' : ''
27: }
28:
29: console.log('Iniciant execució...')
30: setTimeout( function(){
31:    fibo_back2(40, show_back)
32:    fibo_back2('pep', show_back)
33: }, 2000 )
34: console.log('llançat càlcul fibonacci...')
35: setTimeout( function(){
36:    fact_back2(10, show_back)
37:    fact_back2('ana', show_back)
38: }, 1000 )
39: console.log('llançat càlcul factorial...')
```

Ara, les funcions asincròniques, ***fibo_back2*** i ***fact_back2***, reben com callback la funció ***show_back*** (implementada en les línies 19 a 22). S'ha afegit una altra funció auxiliar, ***eval_err*** (en les línies 24 a 27), que genera el missatge d'error quan correspon (s'ha suposat que les situacions d'error es donen quan les funcions s'invoquen amb un argument no numèric).

L'eixida obtinguda en executar el programa ***fibo3.js*** és:

```
Iniciant execució...
```

```
llançat càlcul fibonacci...
llançat càlcul factorial...
factorial(10) = 3628800
factorial(ana) ??? : ana is not a number
fibonacci(40) = 165580141
fibonacci(pep) ??? : pep is not a number
```

Les funcions “asincròniques” definides fins al moment ho són només formalment ja que reben un callback com a argument, però no ho són en el seu comportament: l'asincronia es deu a la seua invocació com a argument de la funció ***setTimeout***.

Una forma de generar versions realment asincròniques consisteix a utilitzar la funció ***nextTick*** de ***process***[3]. Aquesta funció permet retardar l'execució d'una acció fins a la següent iteració del bucle d'esdeveniments. Considere's la següent modificació del programa tenint en compte això:

```
 1: // ***  fibo4.js
 2:
 3: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 4:
 5: function fibo_async(n,cb) {
 6:    process.nextTick(function(){
 7:       var err = eval_err(n,'fibonacci')
 8:       var m   = err ? '': fibo(n)
 9:       cb(err,'fibonacci('+n+') = '+m)
10:    });
11: };
12:
13: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
14:
15: function fact_async(n,cb) {
16:    process.nextTick(function(){
17:       var err = eval_err(n,'factorial')
18:       var m   = err ? '': fact(n)
19:       cb(err,'factorial('+n+') = '+m)
20:    });
21: }
22:
23: function show_back(err,res) {
24:    if (err) console.log(err)
25:    else     console.log(res)
26: }
27:
28: function eval_err(n,s) {
29:    return (typeof n != 'number') ?
30:              s+'('+n+') ??? : '+n+' is not a number' : ''
```

[3] Understanding process.nextTick() - Kishore Nallan: http://howtonode.org/understanding-process-next-tick

```
31: }
32:
33: console.log('Iniciant execució...')
34: fact_async(10, show_back)
35: fact_async('ana', show_back)
36: console.log('llançats càlcul factorial...')
37: fibo_async(40, show_back)
38: fibo_async('pep', show_back)
39: console.log('llançats càlcul fibonacci...')
```

L'eixida obtinguda en executar el programa ***fibo4.js*** és la mateixa que en executar ***fibo3.js***. Observe's, en les línies 33 a 39, que no es requereix l'ús de ***setTimeout***. Observe's també que s'ha modificat l'ordre de les instruccions: les invocacions a ***fact_async*** (línies 34-35) es fan abans que les invocacions a ***fibo_async*** (línies 37-38), atès que ara no s'estableix un temporitzador explícit a cada funció, i es vol seguir mostrant abans el resultat del càlcul de factorial.

Supose's ara que es desitjara calcular un cert nombre de valors resultat d'ambdues funcions i que aquests valors s'emmagatzemaren en sengles vectors. Un programa que fera això seria el següent:

```
 1: // *** fibo5.js
 2:
 3: // *** funcions
 4:
 5: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 6:
 7: function fibo_async(n,cb) {
 8:    process.nextTick(function(){
 9:       var err = eval_err(n,'fibonacci')
10:       var m   = err ? '': fibo(n)
11:       cb(err,n,m)
12:    });
13: };
14:
15: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
16:
17: function fact_async(n,cb) {
18:    process.nextTick(function(){
19:       var err = eval_err(n,'factorial')
20:       var m   = err ? '': fact(n)
21:       cb(err,n,m)
22:    });
23: }
24:
25: function show_fibo_back(err,num,res) {
26:    if (err) console.log(err)
27:    else {
28:       console.log('fibonacci('+num+') = '+res)
29:       fibs[num]=res
30:    }
```

```
31: }
32:
33: function show_fact_back(err,num,res) {
34:    if (err) console.log(err)
35:    else {
36:       console.log('factorial('+num+') = '+res)
37:       facts[num]=res
38:    }
39: }
40:
41: function eval_err(n,s) {
42:    return (typeof n != 'number') ?
43:              s+'('+n+') ??? : '+n+' is not a number' : ''
44: }
45:
46: // ***programa principal
47: console.log('Iniciant execució...')
48:
49: var facts = []
50: for (var i=0; i<=10; i++)
51:    fact_async(i, show_fibo_back)
52: console.log('llançat càlcul de factorials...')
53:
54: var fibs = []
55: for (var i=0; i<=20; i++)
56:    fibo_async(i, show_fact_back)
57: console.log('llançat càlcul de fibonaccis...')
```

En aquest programa, s'ha modificat la implementació del callback. Ara hi ha dues funcions callback (una per a cada funció asincrònica), que emmagatzemen el resultat del càlcul en la posició indicada de l'array corresponent, a més de mostrar-ho en consola.

L'eixida obtinguda en executar el programa ***fibo5.js*** (no es mostra l'eixida completa, les línies amb punts suspensius representen l'eixida no reproduïda explícitament) és:

```
Iniciant execució...
llançat càlcul de factorials...
llançat càlcul de fibonaccis...
factorial(0) = 1
factorial(1) = 1
...
factorial(9) = 362880
factorial(10) = 3628800
fibonacci(0) = 1
fibonacci(1) = 1
...
fibonacci(19) = 6765
fibonacci(20) = 10946
```

A continuació, considere's ampliar el programa perquè proporcione la suma de tots els factorials calculats i la suma de tots els termes de Fibonacci calculats. Podria pensar-se que una possible solució seria la següent:

```
 1: // ***  fibo6a.js
 2:
 3: // *** funcions:  mateixa implementació que en el programa fibo5.js
 4: function fibo(n) { … }
 5: function fibo_async(n,cb) { … }
 6: function fact(n) { … }
 7: function fact_async(n,cb) { … }
 8: function show_fibo_back(err,num,res) { … }
 9: function show_fact_back(err,num,res) { … }
10: function eval_err(n,s) { … }
11:
12: function suma(a,b) { return a+b }
13:
14: // *** programa principal
15: console.log('Iniciant execució...')
16:
17: var n = 10
18: var facts = []
19: var fibs = []
20:
21: for (var i=0; i<n; i++)
22:    fact_async(i, show_fact_back)
23: console.log('llançat càlcul de factorials...')
24:
25: for (var i=0; i<n; i++)
26:    fibo_async(i, show_fibo_back)
27: console.log('llançat càlcul de fibonaccis...')
28:
29: console.log('suma factorials ['+0+'..'+(n-1)+'] =', facts.reduce(suma))
30: console.log('suma fibonaccis ['+0+'..'+(n-1)+'] =', fibs.reduce(suma))
```

No obstant això, l'eixida obtinguda en executar ***fibo6a.js*** és:

```
Iniciant execució...
llançat càlcul de factorials...
llançat càlcul de fibonaccis...
…
… /fibo6a.js:29
   console.log('suma factorials ['+0+'..'+(n-1)+'] =', facts.reduce(suma))

TypeError: Redueix of empty array with no initial value
   at Array.reduce (native)
   …
```

Aquest resultat (error en intentar aplicar la funció ***reduce*** a un vector buit, no inicialitzat) es produeix perquè les línies 29 i 30 s'executen abans que les funcions asincròniques i les seues callbacks.

Una solució senzilla a aquest problema és reorganitzar les instruccions de les línies 29 i 30 amb una funció que passe la seua execució a la cua de torns de Node. Per exemple:

```
process.nextTick( function(){
   console.log('suma factorials ['+0+'..'+(n-1)+'] =', facts.reduce(suma))
   console.log('suma fibonaccis ['+0+'..'+(n-1)+'] =', fibs.reduce(suma))
});
```

O també mitjançant:

```
setTimeout( function(){
   console.log('suma factorials ['+0+'..'+(n-1)+'] =', facts.reduce(suma))
   console.log('suma fibonaccis ['+0+'..'+(n-1)+'] =', fibs.reduce(suma))
}, 1 );
```

Qualsevol d'aquestes dues solucions, proporciona la següent eixida en executar el programa:

```
Iniciant execució...
llançat càlcul de factorials...
llançat càlcul de fibonaccis...
factorial(0) = 1
factorial(1) = 1
...
factorial(9) = 362880
fibonacci(0) = 1
fibonacci(1) = 1
...
fibonacci(9) = 55
suma factorials [0..9] = 409114
suma fibonaccis [0..9] = 143
```

Una altra possibilitat és modificar els callbacks de les funcions asincròniques perquè, una vegada calculades totes les components del vector, calculen les sumes. Aquesta solució seria la següent:

```
1: // ***  fibo6b.js
2:
3: // *** funcions:  mateixa implementació que en fibo6a.js, excepte callbacks
4: function fibo(n) { … }
5: function fibo_async(n,cb) { … }
6: function fact(n) { … }
7: function fact_async(n,cb) { … }
8: function eval_err(n,s) { … }
9: function suma(a,b) { … }
```

```
10:
11: function show_fibo_back(err,num,res) {
12:    if (err) console.log(err)
13:    else {
14:       console.log('fibonacci('+num+') = '+res)
15:       fibs[num]=res
16:       if (num==n-1) {
17:          var s = fibs.reduce(suma)
18:          console.log('suma fibonaccis ['+0+'..'+(n-1)+'] =', s)
19:       }
20:    }
21: }
22:
23: function show_fact_back(err,num,res) {
24:    if (err) console.log(err)
25:    else {
26:       console.log('factorial('+num+') = '+res)
27:       facts[num]=res
28:       if (num==n-1) {
29:          var s = facts.reduce(suma)
30:          console.log('suma factorials ['+0+'..'+(n-1)+'] =', s)
31:       }
32:    }
33: }
34:
35: // *** programa principal
36: console.log('Iniciant execució...')
37:
38: var n = 10
39: var facts = []
40: var fibs = []
41:
42: for (var i=0; i<n; i++)
43:    fact_async(i, show_fact_back)
44: console.log('llançat càlcul de factorials...')
45:
46: for (var i=0; i<n; i++)
47:    fibo_async(i, show_fibo_back)
48: console.log('llançat càlcul de fibonaccis...')
```

L'eixida obtinguda en executar ***fibo6b.js*** és:

```
Iniciant execució...
llançat càlcul de factorials...
llançat càlcul de fibonaccis...
factorial(0) = 1
factorial(1) = 1
...
factorial(9) = 362880
suma factorials [0..9] = 409114
fibonacci(0) = 1
```

```
fibonacci(1) = 1
...
fibonacci(9) = 55
suma fibonaccis [0..9] = 143
```

Aquesta eixida és correcta i només difereix (respecte a les solucions d'organitzar les crides a ***reduce*** amb ***setTimeout*** o ***nextTick***) en l'ordre en el qual es presenten els resultats en la consola.

3. Promeses

Hi ha una altra manera de construir execucions asincròniques en JavaScript: mitjançant promeses. Encara que durant els darrers anys ha hi hagut múltiples propostes per a introduir promeses en JavaScript, suportades per diferents mòduls que podien incloure's en els programes que escrivíem, l'estàndard ECMAscript 6 només ha acceptat la variant basada en constructors, proporcionant l'objecte Promise per a aquesta finalitat.

Aquesta secció aplica la variant estàndard a l'exemple de càlcul dels termes de la successió de Fibonacci, presentat en la secció anterior.

El programa a utilitzar és el següent:

```
 1: // ***  fibo7.js
 2:
 3: // fibo - sync
 4: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 5:
 6: // fibo - new-promise version
 7: function fibo_promise(n) {
 8:    return new Promise(function(fulfill, reject) {
 9:       if (typeof(n)!='number') reject( n+' is an incorrect argument' )
10:          // Unfortunately, fibo() is a synchronous function. We should
11:          // convert it into something apparently "asynchronous". To this end,
12:          // we place its computation in the next scheduler turn.
13:       else    setTimeout( function() {fulfill( fibo(n) )}, 1 )
14:    })
15: }
16:
17: // onFulfilled handler, with closure
18: function onSuccess(i) {
19:    return function (res) { console.log('fibonacci('+i+') =', res) }
20: }
21:
22: // onRejected handler
23: function onError(err) { console.log('Error:', err); }
24:
25: // *** main program
26: console.log('Execution is starting...')
27: var elems = [25, '5', true]
28:
29: for (var i in elems)
30:    fibo_promise(elems[i]).then( onSuccess(elems[i]), onError )
```

La funció ***fibo_promise*** construeix i torna un objecte promesa. Aquesta funció té dos paràmetres. El primer és el callback que cal utilitzar quan la promesa genere un resultat correcte. Com es pot veure en aquest exemple, aquest callback rep com a argument el resultat de la invocació a la funció que es pretén convertir en promesa. En l'exemple ha sigut la funció fibo(). El segon paràmetre és el callback que s'invocarà quan la promesa genere un error o excepció.

Tant la segona com la tercera component del vector “elems” generen un error en invocar fibo(), mostrant-lo de seguida. D’altra banda, fibo(25) necessita prou temps per completar els seus càlculs. Per això, en l’eixida del programa anterior s’observarà primer el missatge d’error per a la cadena “5”, seguit per un missatge similar per al valor booleà “true”. Per a concloure es mostrarà el resultat de fibo(25).

En el següent programa es considera de nou el problema de calcular factorials i termes de Fibonacci emmagatzemant-los en vectors. Una implementació usant promeses és la següent:

```
 1: // ***  fibo8.js
 2:
 3: // fibo - sync
 4: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 5:
 6: // fibo - promise version
 7: function fibo_promise(n) {
 8:    return new Promise(function(fulfill, reject) {
 9:       if (typeof(n)!='number') reject( n+' is an incorrect argument' )
10:       else     setTimeout( function() {fulfill( fibo(n) )}, 1 )
11:    })
12: }
13:
14: // fact - sync
15: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
16:
17: // fact - promise version
18: function fact_promise(n) {
19:   return new Promise(function(fulfill, reject) {
20:      if (typeof(n)== 'number' ) setTimeout( function() {fulfill( fact(n) )}, 1 )
21:      else   reject( n+' is an incorrect arg' )
22:   })
23: }
24:
25: // onFulfilled handler, with closure
26: function onSuccess(s, x, i) {
27:    return function (res) {
28:       if ( x!=null ) x[i] = res
29:       console.log(s+'('+i+') =', res)
30:    }
31: }
32:
33: // onRejected handler
34: function onError(err) { console.log('Error:', err); }
35:
36: // *** main program
37: console.log('Execution is starting...')
38:
39: var n = 10
40: var fibs = []
41: var fibsPromises = []
42: var facts = []
```

```
43: var factsPromises = []
44:
45: // Generate the promises.
46: for (var i=0; i<n; i++) {
47:    fibsPromises[i] = fibo_promise(i)
48:    factsPromises[i] = fact_promise(i)
49: }
50:
51: // Show the results.
52: for (var i=0; i<n; i++) {
53:    fibsPromises[i].then( onSuccess('fibonacci', fibs, i), onError )
54:    factsPromises[i].then( onSuccess('factorial', facts, i), onError )
55: }
```

L'eixida obtinguda en executar el programa ***fibo8.js*** és:

```
Execution is starting...
fibonacci(0) = 1
factorial(0) = 1
…
fibonacci(9) = 55
factorial(9) = 362880
```

Considere's ara el mateix problema, i a més el problema de sumar les components dels vectors resultat. Un programa correcte usant promeses és el següent:

```
 1: // ***  fibo9.js
 2:
 3: // fibo - sync
 4: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 5:
 6: // fibo - promise version
 7: function fibo_promise(n) {
 8:    return new Promise(function(fulfill, reject) {
 9:       if (typeof(n)!='number') reject( n+' is an incorrect argument' )
10:       else    setTimeout( function() {fulfill( fibo(n) )}, 1 )
11:    })
12: }
13:
14: // fact - sync
15: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
16:
17: // fact - promise version
18: function fact_promise(n) {
19:   return new Promise(function(fulfill, reject) {
20:      if (typeof(n)== 'number' ) setTimeout( function() {fulfill( fact(n) )}, 1 )
21:      else   reject( n+' is an incorrect arg' )
22:   })
23: }
24:
```

```
25: // onFulfilled handler, with closure
26: function onSuccess(s, x, i) {
27:    return function (res) {
28:       if ( x!=null ) x[i] = res
29:       console.log(s+'('+i+') =', res)
30:    }
31: }
32:
33: // onRejected handler
34: function onError(err) { console.log('Error:', err); }
35:
36: // onFulfilled handler, for array of promises
37: function sumAll(z, x) {
38:    return function () {
39:       var s = 0
40:       for (var i in x) s += x[i]
41:       console.log(z, '=', x, '; sum =', s)
42:    }
43: }
44:
45: // onRejected handler, for array of promises
46: function showFinalError() {
47:    console.log('Something wrong has happened...')
48: }
49:
50: // *** main program
51: console.log('Execution is starting...')
52:
53: var n = 10
54: var fibs = []
55: var fibsPromises = []
56: var facts = []
57: var factsPromises = []
58:
59: // Generate the promises.
60: for (var i=0; i<n; i++) {
61:    fibsPromises[i] = fibo_promise(i)
62:    factsPromises[i] = fact_promise(i)
63: }
64:
65: // Show the results.
66: for (var i=0; i<n; i++) {
67:    fibsPromises[i].then( onSuccess('fibonacci', fibs, i), onError )
68:    factsPromises[i].then( onSuccess('factorial', facts, i), onError )
69: }
70: // Show the summary.
71: Promise.all(fibsPromises).then( sumAll('fibs', fibs), showFinalError )
72: Promise.all(factsPromises).then( sumAll('facts', facts) )
```

L'eixida obtinguda en executar el programa ***fibo9.js*** és:

```
Execution is starting...
fibonacci(0) = 1
factorial(0) = 1
…
fibonacci(9) = 55
factorial(9) = 362880
fibs = [ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ] ; sum = 143
facts = [ 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880 ] ; sum = 409114
```

Com s'acaba de comprovar, l'ús de vectors de promeses resol satisfactòriament el problema de calcular la suma de components d'un vector. El manejador d'èxit associat al resultat del vector de promeses, funció ***sumAll***, s'executa solament després que totes les promeses han sigut satisfetes, és a dir, després que s'hagen calculat tots els valors que es desitjava emmagatzemar al vector.

L'encadenament de les promeses usant la funció ***then*** facilita al programador establir la seqüència adequada d'accions. Com es pot apreciar, aquesta seqüència és més fàcil de llegir en el codi (respecte a la solució basada en callbacks).

En els programes ***fibo8.js*** i ***fibo9.js***, es pot apreciar l'extrema similitud de les funcions ***fibo_promise*** i ***fact_promise***. En el següent programa es proporciona una altra implementació equivalent, amb menys codi:

```
 1: // ***  fibo10.js
 2:
 3: // fibo - sync
 4: function fibo(n) { return (n<2) ? 1 : fibo(n-2) + fibo(n-1) }
 5:
 6: // fact - sync
 7: function fact(n) { return (n<2) ? 1 : n * fact(n-1) }
 8:
 9: // "func" - promise version
10: function func_promise(f,n) {
11:    return new Promise(function(fulfill, reject) {
12:       if (typeof(n)!='number') reject( n+' is an incorrect argument' )
13:       else    setTimeout( function() {fulfill( f(n) )}, 1 )
14:    })
15: }
16:
17: // onFulfilled handler, with closure
18: function onSuccess(s, x, i) {
19:    return function (res) {
20:       if ( x!=null ) x[i] = res
21:       console.log(s+'('+i+') =', res)
22:    }
23: }
24:
25: // onRejected handler
26: function onError(err) { console.log('Error:', err); }
27:
```

```
28: // onFulfilled handler, for array of promises
29: function sumAll(z, x) {
30:    return function () {
31:       var s = 0
32:       for (var i in x) s += x[i]
33:       console.log(z, '=', x, '; sum =', s)
34:    }
35: }
36:
37: // onRejected handler, for array of promises
38: function showFinalError() {
39:    console.log('Something wrong has happened...')
40: }
41:
42: // *** main program
43: console.log('Execution is starting...')
44:
45: var n = 10
46: var fibs = []
47: var fibsPromises = []
48: var facts = []
49: var factsPromises = []
50:
51: // Generate the promises.
52: for (var i=0; i<n; i++) {
53:    fibsPromises[i]  = func_promise(fibo, i)
54:    factsPromises[i] = func_promise(fact, i)
55: }
56:
57: // Show the results.
58: for (var i=0; i<n; i++) {
59:    fibsPromises[i].then( onSuccess('fibonacci', fibs, i), onError )
60:    factsPromises[i].then( onSuccess('factorial', facts, i), onError )
61: }
62:
63: // Show the summary.
64: Promise.all(fibsPromises).then( sumAll('fibs', fibs), showFinalError )
65: Promise.all(factsPromises).then( sumAll('facts', facts) )
```