Programación Asíncrona en Node JS

Modelo de Paso de Continuaciones, Eventos, Promesas y Generadores

Javier Vélez Reyes

@javiervelezreye
Javier.velez.reyes@gmail.com

Mayo 2014

I. ¿Quién Soy?

Licenciado en informática por la Universidad Politécnica de Madrid (UPM) desde el año 2001 y doctor en informática por la Universidad Nacional de Educación a Distancia (UNED) desde el año 2009, Javier es investigador y está especializado en el diseño y análisis de la colaboración. Esta labor la compagina con actividades de evangelización, consultoría, mentoring y formación especializada para empresas dentro del sector IT. Inquieto, ávido lector y seguidor cercano de las innovaciones en tecnología.

II. ¿A Qué Me Dedico?

E-learning
Diseño de Sistemas de Colaboración
Learning Analytics
Gamificación Colaborativa
Diseño Instruccional

Desarrollado Front/Back
Evangelización Web
Arquitectura Software
Formación & Consultoría IT

Javier Vélez Reyes
@javiervelezreye
Javier.veler.reyes@gmail.com

1 Introducción

- Programación Secuencial
- Programación Asíncrona
- Modelos de Programación Asíncrona
- Un ejemplo

I. Programación Secuencial

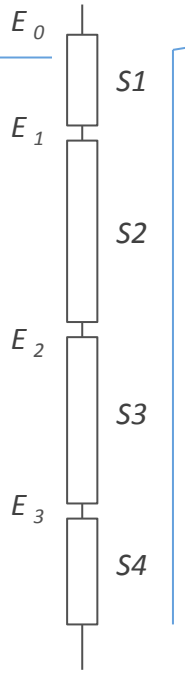
Tradicionalmente, el modelo de ejecución que utilizan la mayoría de los lenguajes y paradigmas de programación se corresponde con un tratamiento secuencial del cuerpo del programa. Un programa es entendido como una secuencia de instrucciones que se ejecutan ordenadamente y donde la ejecución de cada operación no da comienzo hasta que no termina la anterior.

Operaciones bloqueantes

*Se dice que las operaciones son **bloqueantes** ya que bloquean el flujo de ejecución del programa llamante impidiendo que la siguiente instrucción comience hasta que la anterior haya terminado*

Razonamiento basado en estado

*Con este esquema resulta fácil razonar acerca del comportamiento del programa si éste se concibe como un proceso de **transformación de estado** reflejado en las variables a lo largo del tiempo*



Ejecución secuencial

*El programa se ejecuta de principio a fin respetando el **orden** correlativo que cada instrucción guarda con la instrucción anterior y siguiente*

II. Programación Asíncrona

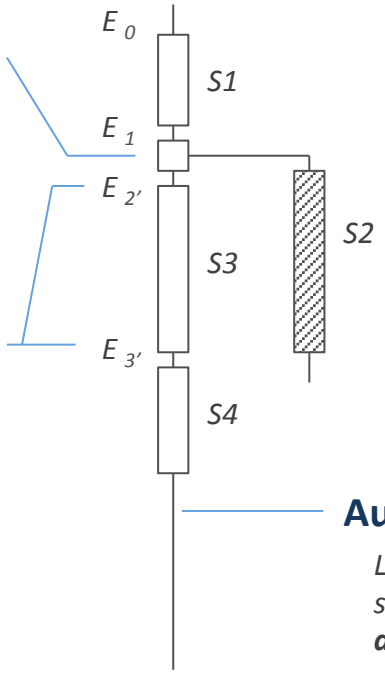
La programación asíncrona establece la posibilidad de hacer que algunas operaciones devuelvan el control al programa llamante antes de que hayan terminado mientras siguen operando en segundo plano. Esto agiliza el proceso de ejecución y en general permite aumentar la escalabilidad pero complica el razonamiento sobre el programa.

Operaciones no bloqueantes

*Ahora algunas operaciones son **no bloqueantes** ya que devuelven el control al programa llamante antes de su terminación mientras siguen ejecutando en segundo plano*

Imposible razonar sobre el estado

La falta de secuencialidad estricta en la ejecución del programa hace difícil determinar el estado al comienzo de cada operación



Ejecución no secuencial

*Ya **no** se mantiene el **orden** secuencial puesto la ejecución de la instrucción que sigue a un operación no bloqueante se adelanta antes de que dicha no bloqueante haya finalizado*

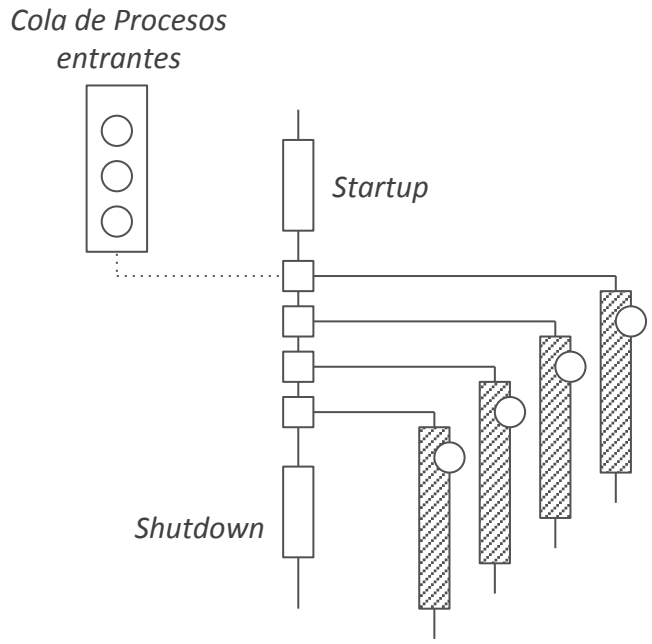
Aumento de la escalabilidad

*La asincronía permite que los siguientes procesos en espera **adelanten** su ejecución*

II. Programación Asíncrona

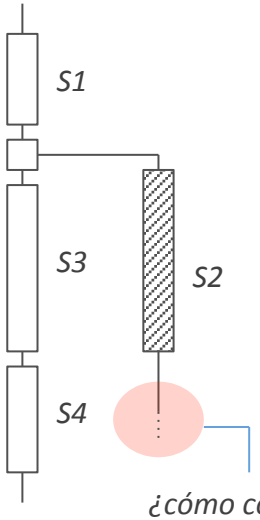
Lo Bueno

La programación asíncrona resulta ventajosa ya que aumenta el throughput soportado. Esta ventaja le está haciendo coger mucha tracción dada la demanda de sistemas de alta escalabilidad que se requieren en Internet.



Lo Malo

El principal problema de la programación asíncrona se refiere a cómo dar continuidad a las operaciones no bloqueantes del algoritmo una vez que éstas han terminado su ejecución.



¿cómo continuamos? ¿cómo recuperamos el estado en este punto? ¿cómo obtengo los resultados?

III. Modelos de Programación Asíncrona

Para dar respuesta al problema anterior – cómo dar tratamiento de continuidad al resultado de las operaciones no bloqueantes una vez que éstas han terminado – se han establecido diferentes modelos de programación asíncrona. Las bondades de dichos modelos se valoran en términos de cuánto permiten aproximar la programación asíncrona a un esquema lo más parecido al secuencial.

I. Modelo de Paso de Continuaciones

Es el modelo de asincronía definido dentro de Node JS. Cada función recibe información acerca de cómo debe tratar el resultado – de éxito o error – de cada operación. Requiere orden superior

II. Modelo de Eventos

Se utiliza una arquitectura dirigida por eventos que permite a las operaciones no bloqueantes informar de su terminación mediantes señales de éxito o fracaso. Requiere correlación para sincronizar

III. Modelo de Promesas

Se razona con los valores de retorno de las operaciones no bloqueantes de manera independiente del momento del tiempo en que dichos valores – de éxito o fallo – se obtengan

IV. Modelo de Generadores

Se utilizan generadores para devolver temporalmente el control al programa llamante y retornar en un momento posterior a la rutina restaurando el estado en el punto que se abandonó su ejecución

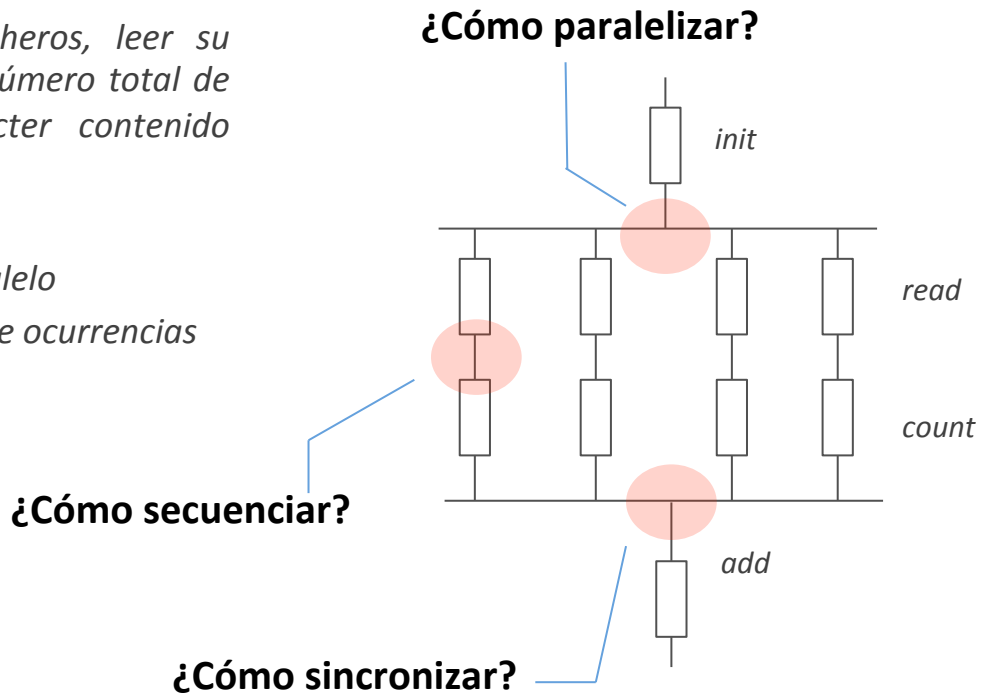
IV. Un ejemplo

Sequential.js

Para entender comparativamente cada uno de estos cuatro modelos utilizaremos a lo largo de esta charla un sencillo ejemplo que expone los 3 tipos de problemas característicos relacionados con el control de flujo dentro de programas asíncronos.

Dada una colección de ficheros, leer su contenido y contabilizar el número total de ocurrencias de cada carácter contenido dentro de los mismos

1. *Leer cada fichero en paralelo*
2. *Contabilizar su número de ocurrencias*
3. *Sumar los resultados*



Javier Vélez Reyes
@javiervelezreye
Javier.veler.reyes@gmail.com

2

Node JS como Lenguaje Asíncrono

- La Asincronía de Node JS
- Principios Arquitectónicos de Node JS
- Arquitectura de Node JS

I. La Asincronía de Node JS

Parece natural pensar que la programación asíncrona exige de un contexto multi-hilo ya que el carácter no bloqueante se consigue por medio de una suerte de ejecución simultanea que transcurre en un segundo plano del flujo principal del programa. Sin embargo, esto no implica necesariamente un contexto de ejecución concurrente ya que las operaciones no bloqueantes pueden ejecutarse de forma aislada.

> Node JS is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node JS uses an **event-driven**, **non-blocking I/O** model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Modelo no bloqueante de E/S

Node JS es un lenguaje single-thread pero que aplica multi-threading en los procesos de entrada salida y es ahí donde se aplica el carácter no bloqueante

Arquitectura dirigida por eventos

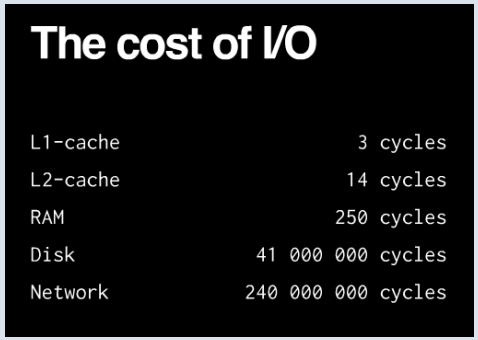
Node JS utiliza alternativamente, como veremos, el modelo de paso de continuaciones y el de eventos si bien su arquitectura general está dirigida por un loop general de eventos

II. Principios Arquitectónicos de Node JS

A pesar de que recientemente Node JS está recibiendo, especialmente desde ciertas comunidades competidoras, fuertes críticas relativas a su aprovechamiento de los ciclos de cómputo por tratarse de un entorno single-thread, su filosofía se basa en tres fuertes principios arquitectónicos.

1. La E/S es lo que más coste implica

Esta experimentalmente comprobado que procesamiento de las operaciones de E/S es el que mayor coste implica dentro de las arquitecturas Hw.



2. Dedicar un hilo por solicitud es caso

Dedicar un hilo para enhebrar el procesamiento de cada solicitud entrante, como hacen otras arquitecturas servidoras (Apache) resulta demasiado caro en memoria

3. Todo en paralelo menos tu código

Como consecuencia el esquema de comportamiento de Node JS se puede resumir en aquellas partes del proceso de la petición que merezca la pena paralelizar (E/S) se ejecutarán de forma no bloqueante mientras que el resto ejecuta en un esquema single-thread

III. Arquitectura de Node JS

1. Nueva Solicitud

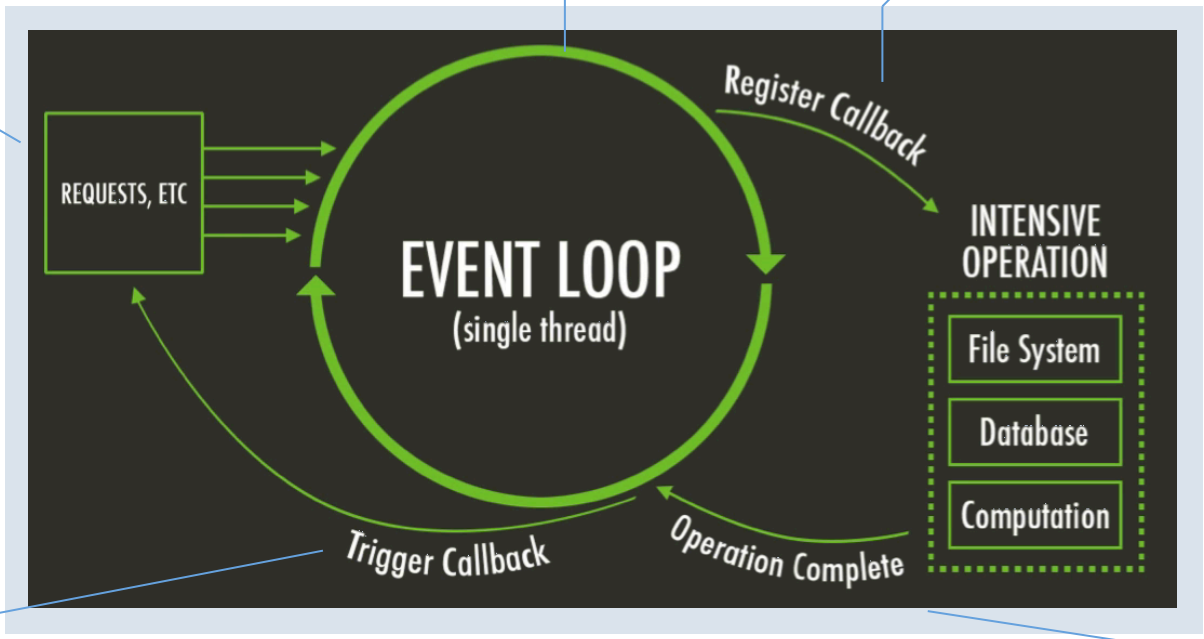
Llegan las solicitudes a Node JS desde el exterior

2. Gestión

Un único hilo se encarga de gestionar las solicitudes entrantes

3. Ejecución no bloquante

Las operaciones de E/S se procesan en paralelo en modo no bloqueante



Todo Se Ejecuta en Paralelo menos tu Código

5. Procesar resultados

Se ejecuta la lógica de continuidad que procesa los resultados

4. Finaliza la operación

Cuando la operación termina vuelve al loop

Javier Vélez Reyes
@javiervelezreye
Javier.veler.reyes@gmail.com

3

Modelo de Paso de Continuidades

- Qué es una Continuación
- Control de Flujo Mediante Continuaciones
- Ejemplo
- Librerías
- Conclusiones

I. Qué es Una Continuación

El modelo nativo que utiliza Node JS en sus APIs para dar soporte a la programación asíncrona es el de paso de continuaciones. Cada operación no bloqueante recibe una función como último parámetro que incluye la lógica de continuación que debe ser invocada tras la finalización de la misma tanto para procesar los resultados en caso de éxito como para tratar los fallos en caso de error.

Proveedor

```
function div(a, b, callback) {
   if (b === 0) callback (Error (...))
   else {
     var result = a / b;
     callback (null, result);
   }
}
```

Paso de Continuación

La función de continuación permite indicar a la operación bloqueante como debe proceder después de finalizada la operación

Cliente

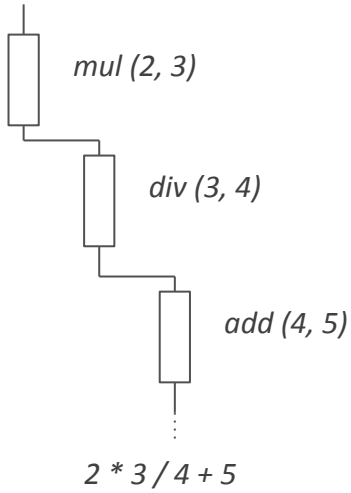
```
div (8, 2, function (error, data) {
  if (error) console.error (error);
  else console.log (data);
});
```

II. Control de Flujo Mediante Continuaciones

A. Secuenciamiento

La manera de proceder dentro de este modelo para establecer flujos de ejecución secuenciales exige ir encadenando cada función subsiguiente como continuación de la anterior donde se procesarán los resultados tanto en caso de éxito como de fracaso. Esto conduce a una diagonalización del código que se ha dado en llamar pirámide del infierno (callback hell), por su falta de manejabilidad práctica en cuanto crece mínimamente el número de encadenamientos secuenciales.

```
mul(2, 3, function (error, data) {
  if (error) console.error (error);
  else div(data, 4, function (error, data) {
    if (error) console.error (error);
    else add(data, 5, function (error, data) {
      ...
    });
  });
});
```

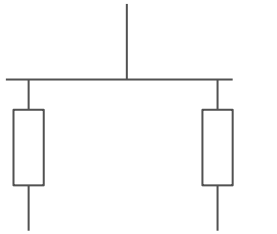


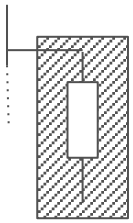
II. Control de Flujo Mediante Continuaciones

B. Paralelización

La paralelización – ejecución asíncrona – de las operaciones no bloqueantes es inmediata ya que su mera invocación ejecuta en segundo plano por definición. Para convertir en no bloqueantes las operaciones bloqueantes, se requiere un pequeño proceso de encapsulación funcional que lance la operación en segundo plano.

```
add(2, 3, function (error, data) {...});
sub(3, 4, function (error, data) {...});
```





```
function doAsync (fn, callback, self) {
  return function () {
    var allParams = [].slice.call(arguments);
    var params    = allParams.slice (0, allParams.length - 1);
    var callback  = allParams[allParams.length - 1];
    setTimeout (function () {
      var results = fn.apply (self, params)
      callback (null, results);
    }, 0);
  };
}
```

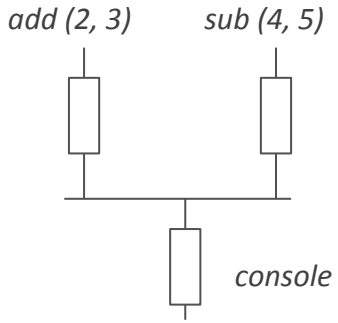
II. Control de Flujo Mediante Continuaciones

C. Sincronización

La sincronización de funciones de continuación requiere encadenar al final de cada secuencia paralela una función de terminación que aplique cierta lógica sólo una vez que se compruebe que todas los ramas paralelas han terminado. Para implementar esta comprobación se utilizan esquemas basados en contadores.

```
function doParallel(fns, endCallback, params) {
  var pending  = fns.length;
  var callback = function (error, data) {
    <<processing data && error>>
    pending --;
    if (pending === 0) {
      endCallback();
    }
  }
  for (var index = 0; index <fns.length; index ++) {
    fns[index].apply (this, params[index], callback);
  }
}
```

```
doParallel ([add, sub], function (error, data) {
   console.log (data)
}, [[2,3],[4,5]]);
```



III. Ejemplo

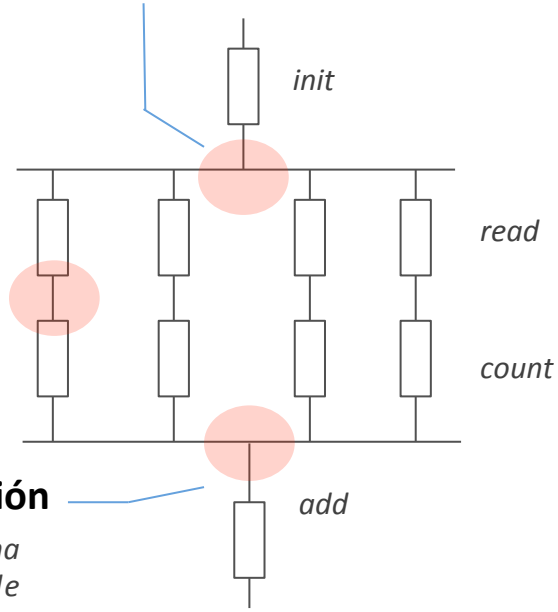
Continuations.js

Dada una colección de ficheros, leer su contenido y contabilizar el número total de ocurrencias de cada carácter contenido dentro de los mismos

1. *Leer cada fichero en paralelo*
2. *Contabilizar su número de ocurrencias*
3. *Sumar los resultados*

Paralelización

Un bucle permite lanzar en ejecución todos los pares no bloqueantes de leer-contar de forma no bloqueante



Secuenciamiento

Cada par leer-contar se encadena a través del paso de funciones de continuación

Sincronización

Para sincronizar cada rama paralela recibe una última continuación que ejecuta lógica de terminación una vez que se ha asegurado que todas las ramas han terminado

IV. Librerías

Existen numerosas librerías que pueden ayudarnos a hacer la vida más fácil cuando trabajamos con un modelo de programación asíncrona basado en continuaciones. Algunas de ellas no están estrictamente relacionadas con la programación asíncrona sino con el paradigma funcional lo cual se debe a que los mecanismos de inyección de continuaciones son en esencia artificios funcionales.

Async

Async es tal vez la librería más conocida y ampliamente utilizada para la programación asíncrona basada en continuaciones. Ofrece métodos de control de flujo variados para funciones no bloqueantes

Join

Join es una implementación del método de sincronización que hemos comentado anteriormente y resulta similar al que puede encontrarse en otros lenguajes como C que opera con threads. También puede usarse en promesas aunque su uso resulta menos relevante

Fn.js

Fn.js es una excelente librería que implementa distintos métodos de gestión funcional. Su aplicabilidad práctica en este contexto está relacionada con las capacidades que expone para generar funciones no bloqueantes y aplicar currificación

V. Conclusiones

Lo Bueno

La programación asíncrona basada en continuidades puede ser una buena opción para situaciones en que la lógica de control de flujo es muy sencilla. Tal suele ser el caso de programas en Node JS que permiten definir una respuesta no bloqueante a peticiones entrantes

Sencillo para esquemas solicitud / respuesta

Alineado con los esquemas de programación funcional

Fácil de entender como mecanismo conceptual

Lo Malo

No obstante, cuando la lógica de control resulta mínimamente elaborada el proceso de razonamiento sobre el programa se complica lo cual redunda en un código con lógica funcional distribuida y difícil de leer, entender y mantener.

Complejidad en la definición de una lógica de control de flujo elaborada

Difícil establecer mecanismos de sincronización

La lógica de control queda distribuida entre cada rama no bloqueante

Difícil de seguir, leer y mantener a medida que crece el código

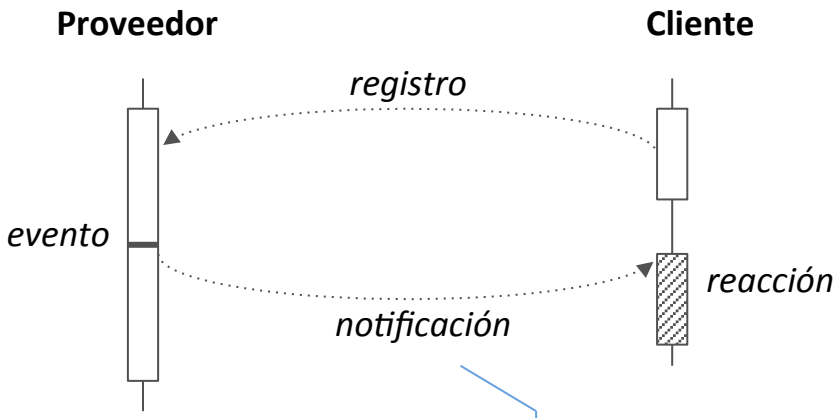
Javier Vélez Reyes
@javiervelezreye
Javier.veler.reyes@gmail.com

4 Modelo de Eventos

- Qué es una Arquitectura Dirigida por Eventos
- Control de Flujo Mediante Eventos
- Ejemplo
- Librerías
- Conclusiones

I. Qué es Una Arquitectura dirigida por Eventos

Un evento es la señalización de un acontecimiento relevante dentro del ecosistema de negocio. Anatómicamente están formados, típicamente, por un tipo, una marca de tiempo y un conjunto de datos que caracteriza el contexto en el que se produjo el evento. Las arquitecturas de eventos (EDA) proporcionan un mecanismo de comunicación entre clientes y proveedores en relación 1:N y con desacoplamiento nominal. Una de sus muchas aplicaciones es en los problemas de procesamiento asíncrono de datos.



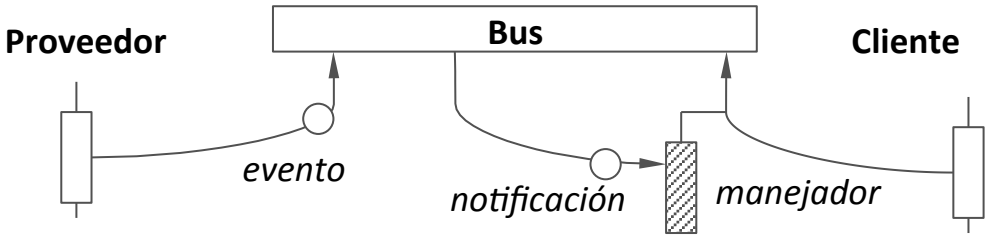
Arquitectura de Eventos

Se produce una comunicación desacoplada entre proveedores y clientes a través de un mecanismo de registro/notificaciones. Nótese que en este esquema las flechas no indican invocaciones sino comunicación indirecta

I. Qué es Una Arquitectura dirigida por Eventos

Arquitectura Centralizada

En las arquitecturas centralizadas dirigidas por eventos existe un mediador central – bus de comunicaciones – que se encarga de hacer efectivo el proceso de registro de los clientes escuchadores y de lanzar las notificaciones bajo demanda de los proveedores a los mismos. Este mecanismo permite una cardinalidad N:N. Este esquema se conoce bajo el nombre de patrón **PUB/SUB**.



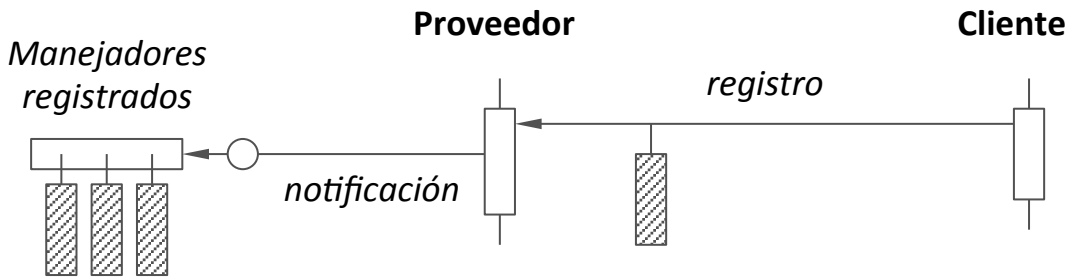

```
var bus = Bus.create ();
bus.send('app.module.event', data);
```

```
var bus = Bus.create ();
bus.receive('app.module.event',
            function (data) {
              ...
            });
bus.refuse('app.module.event');
bus.refuseAll();
```

I. Qué es Una Arquitectura dirigida por Eventos

Arquitectura Distribuida

En las arquitecturas distribuidas dirigidas por eventos cada proveedores es responsable de gestionar la suscripción de sus clientes y de enviar las notificaciones cuando se produce un evento. El mecanismo de comunicación también es desacoplado nominalmente pero la cardinalidad típicamente es de 1:N entre el proveedor y los clientes. Este esquema se corresponde con el patrón **observador-observable** o **event emitters** en jerga Node JS.




```
var events = require('events');
var emitter = new events.EventEmitter();
emitter.emit('app.module.event', data);
```

```
emmiter.on('app.module.event',
            function h(data) {
               ...
            });
bus.removeListener('app.module.event', h);
bus.removeAllListeners('app.module.event');
```

II. Control de Flujo Mediante Eventos

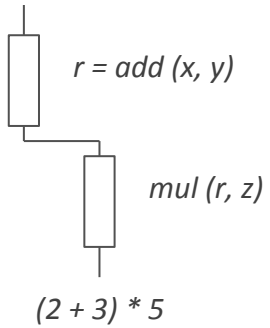
A. Secuenciamiento

El secuenciamiento dentro de este modelo se consigue a través del encadenamiento de escuchadores. Los resultados de las operaciones van cayendo en cascada desde la primera hasta la última acumulándose parcialmente hasta llegar a un resultado final que es recogido y procesado por el cliente.

```
var addEmitter = new events.EventEmitter();
var mulEmitter = new events.EventEmitter();

function add(x, y) {
   addEmitter.emit('result', x+y);
}
function mul(z) {
   addEmitter.on('result', function (data){
      mulEmitter.emit('result', data * z);
   });
}

mul(5);
mulEmitter.on('result', function (data){
   console.log(data);
});
add (2,3);
```



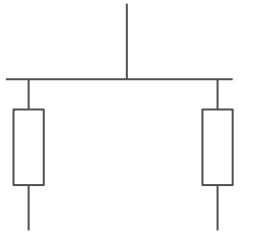
II. Control de Flujo Mediante Eventos

B. Paralelización

En tanto que las arquitecturas dirigidas por eventos establecen un marcado desacoplamiento entre los agentes participantes, lo único que es necesario hacer para paralelizar varias operaciones es hacer que cada una opere de manera independiente y lance eventos cuando genere resultados.

```
var addEmitter = new events.EventEmitter();
var subEmitter = new events.EventEmitter();
function add(data) {
   addEmitter.emit('result', x+y);
}
function sub(x, y) {
   subEmitter.emit('result', x-y);
}
```

```
var emitter = new events.EventEmitter();
function add(x, y) {
   emitter.emit('add', x+y);
}
function sub(x, y) {
   emitter.emit('sub', x-y);
}
```

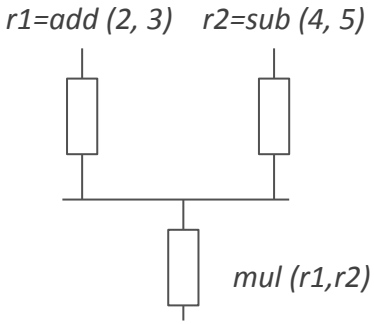


II. Control de Flujo Mediante Eventos

C. Sincronización

Finalmente, la sincronización requiere un proceso de escucha de todas las fuentes emisoras de eventos que operan sobre el final de cada rama paralela. Debe incluirse en las funciones manejadoras lógica de tratamiento que emita nuevos eventos más generales con datos resultantes el proceso de sincronización. A esto se le llama **correlación de eventos**.

```
var addEmitter = new events.EventEmitter();
var subEmitter = new events.EventEmitter();
...
function mul() {
  var r;
  function h(data) {
    if (r) {console.log (r * data)}
    else r = data;
  };
  addEmitter.on ('result', h);
  subEmitter.on ('result', h);
}
mul ();
```



Correlación de Eventos

Existe una gran colección de patrones de diseño propios de las arquitecturas de correlación entre los que se cuentan, transformaciones, agregados, filtros o abstracciones temporales

III. Ejemplo

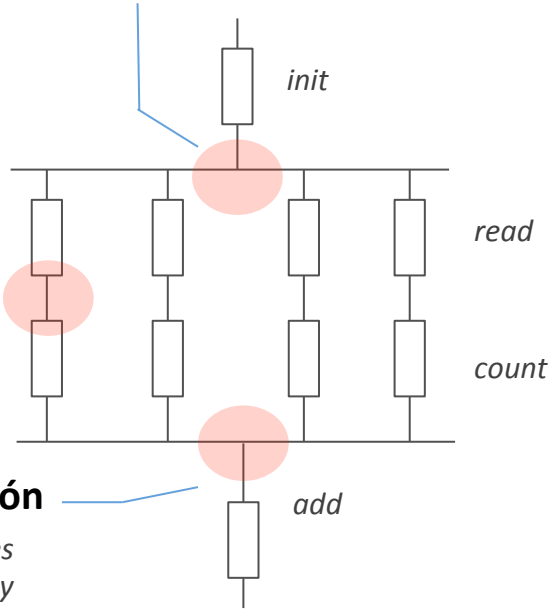
Events.js

Dada una colección de ficheros, leer su contenido y contabilizar el número total de ocurrencias de cada carácter contenido dentro de los mismos

1. *Leer cada fichero en paralelo*
2. *Contabilizar su número de ocurrencias*
3. *Sumar los resultados*

Paralelización

Se lanzan a ejecución de forma iterativa todas las ramas secuenciales para que operen en paralelo emitiendo eventos



Secuenciamiento

La operación read emite eventos de datos leídos del fichero en bloques de 1024 bytes. La operación count escucha esos eventos y genera resultados acumulativamente que emite como evento sólo cuando read envía la señal de fin de fichero

Sincronización

La operación add sincroniza todas las operaciones count. Va calculando los totales acumulativamente y sólo emite un evento cuando determina que se han leído todos los resultados

IV. Librerías

Dado que existen dos estilos arquitectónicos que operan dentro del modelo de las arquitecturas dirigidas por eventos – el centralizado y el distribuido – parece razonable dividir las contribuciones de librerías que existen entre esas dos categorías.

Arquitecturas Distribuidas

Events

Events es un módulo estándar de Node JS utilizado para trabajar con eventos. Dispone de un constructor de emisor de eventos con todos los métodos necesarios para el registro, desregistro y propagación de eventos.

Util

*Algunos autores utilizan el método **inherits** de la librería estándar util con el fin de explotar las capacidades de la librería events por herencia en lugar de por delegación*

Arquitecturas Centralizadas

Postal

Postal es un bus de comunicaciones que implementa el patrón PUB-SUB tanto para cliente como para servidor. Usa expresiones regulares sobre el tipo de eventos para gestionar familias y goza de buena comunidad

V. Conclusiones

Lo Bueno

El modelo dirigido por eventos ofrece grandes posibilidades y da respuesta a un abanico nuevo de problemas que resultan concomitantes con la programación reactiva y las soluciones de correlación de eventos.

Desacoplamiento nominal

Esquemas de comunicación 1:N o N:M

Fácil extensibilidad del sistema reactivo por medio de la adición de nuevos manejadores

Razonamos localmente en problemas desacoplados

Lo Malo

Aunque este modelo es altamente prometedor y supone grandes ventajas con respecto al de paso de continuaciones, no deja de tener detractores que consideran el uso de promesas un artefacto demasiado artificial e incomodo de tratar.

Los procesos de coordinación resultan complicados

La lógica de secuenciamiento queda diluida entre los manejadores

Resulta más invasivo que otras soluciones

Código difícil de seguir, mantener y depurar a medida que crece el tamaño del problema

Javier Vélez Reyes
@javiervelezreye
Javier.veler.reyes@gmail.com

5

Modelo de Promesas

- Qué es una Promesa
- Control de Flujo Mediante Promesas
- Ejemplo
- Librerías
- Conclusiones

I. Qué es Una Promesa

Una promesa es una abstracción computacional que representa un compromiso por parte de la operación no bloqueante invocada de entregar una respuesta al programa llamante cuando se obtenga un resultado tras su finalización. La promesa es un objeto que expone dos métodos inyectores **then** y **fail** para incluir la lógica de tratamiento en caso de éxito o fracaso.

```
var promise = div (a, b);
promise.then (function (data) {
    console.log (data)
}).fail (function (error) {
    console.error (error)
});
```

Inyección de manejadores

*La promesa incluye dos funciones, **then** y **fail**, que permiten indicar cómo tratar los resultados o manejar los errores una vez que la operación no bloqueante ha terminado*

Comunicación basada en promesas

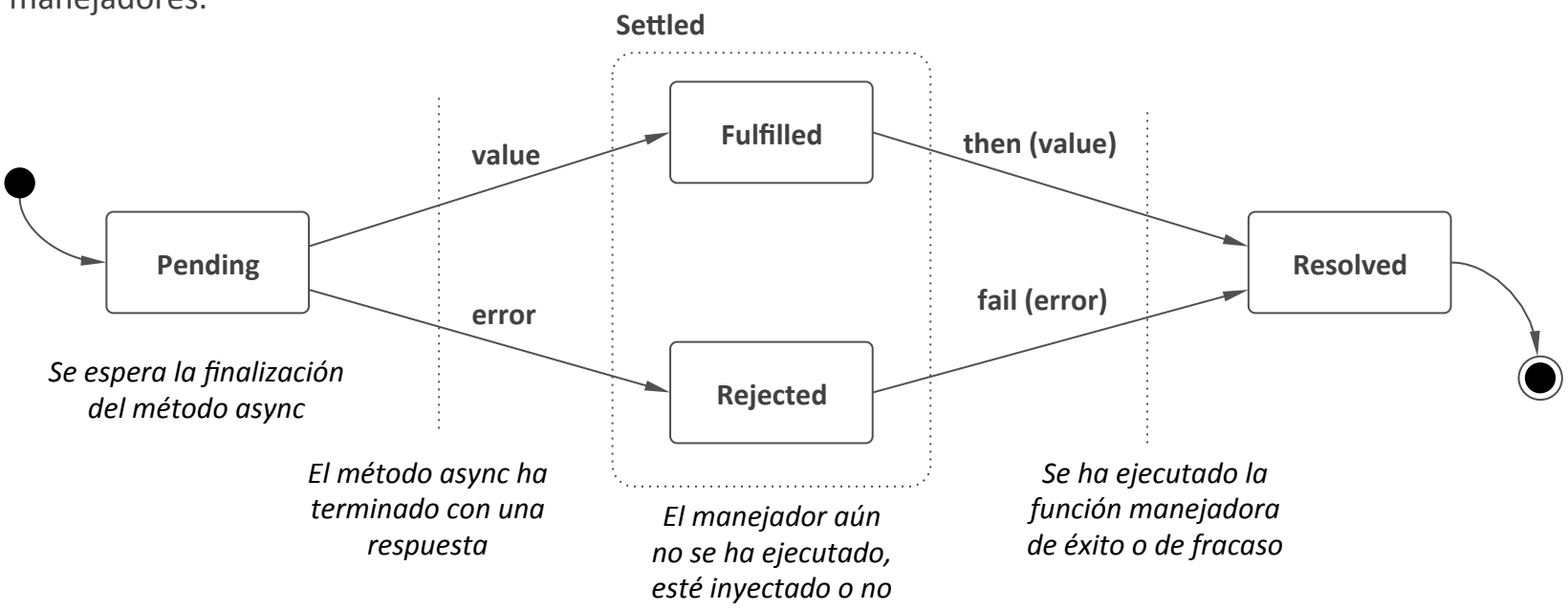
Se recupera el control sobre el flujo del programa. Los proveedores vuelven a devolver (promesas de) valores y los clientes utilizar esas (promesas de) valor en una expresión

```
function div (x, y) {
  var result = ...
  return <<toPromise (result)>>;
}
```

I. Qué es Una Promesa

Ciclo de vida de una promesa

Las promesas responden a un sencillo ciclo de vida que es necesario conocer para operar con ellas convenientemente. El valor esencial de una promesa reside en 2 principios. Primero que la lógica de tratamiento en caso de éxito o fracaso sólo se ejecuta una vez. Y segundo que se garantiza la ejecución de la lógica de éxito o fracaso aunque la promesa se resuelva antes de haber inyectado sus manejadores. La promesa espera, si es necesario a disponer de sus manejadores.



I. Qué es Una Promesa

Construcción de Promesas

Existen diversas formas de obtener promesas que pueden identificarse como patrones de construcción que aparecen recurrentemente al utilizar este modelo. A continuación comentamos los más relevantes con ejemplos en la librería Q.

Obtención directa

```
var promise = getPromise ();
```

Resolución directa

```
var promise = Q.resolve (value);
```

Rechazo directo

```
var promise = Q.reject(error);
```

Denodificación

```
var f = Q.denodeify(fs.readFile);
var f = Q.nfbind (fs.readFile);
Q.nfcall (fs.readFile (...));
```

Invocación funcional

```
return Q.fcall (function () {
    return value;
});
```

Factoría de diferidos

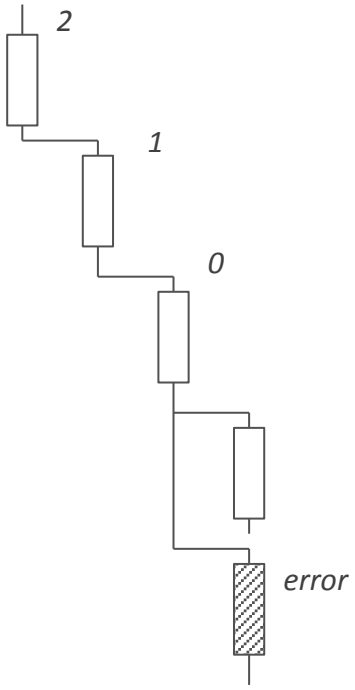
```
var deferred = Q.defer();
if (error) deferred.reject (error);
if (data) deferred.resolve (data);
return deferred.promise;
```

II. Control de Flujo Mediante Promesas

A. Secuenciamiento

La lógica de control secuencial puede obtenerse mediante el encadenamiento the sucesivas invocaciones al método de inyección then. Este encadenamiento provoca un comportamiento secuencial puesto que cada función then genera, al retornar, una promesa que encapsula el valor devuelto (a menos que éste sea ya una promesa) lo que obliga a mantener el orden.

```
Q.resolve (2)
.then (function (value) {
   return value - 1;
})
.then (function (value) {
   return value - 1;
})
.then (function (value) {
   if (value===0) throw Error ();
   return (8 / value);
})
.then (function (value) {
   return value + 1;
})
.fail (function (error) {
   console.log (error);
});
```



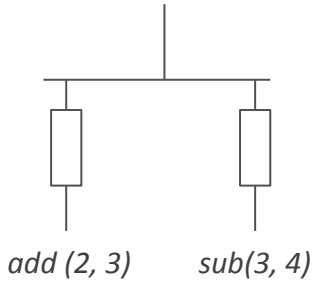
II. Control de Flujo Mediante Promesas

B. Paralelización

Al igual que ocurría en el modelo de paso de continuidades, la paralelización se consigue por invocación directa de las operaciones, ya que éstas tienen un comportamiento no bloqueante. En este modelo sin embargo la programación resulta más natural ya que cada operación devuelve un valor de retorno instantáneamente en forma de promesa.

```
var r1 = add(2, 3);
var r2 = sub(3, 4);
```

```
var r1 = Q.fcall (add, 2, 3);
var r2 = Q.fcall (sub, 3, 4);
```




```
function doParallel(fns, params) {
  var promises = [];
  for (var index = 0; index < fns.length; index ++) {
    var p = Q.fapply (fns[index], params[index]);
    promises.push (p);
  }
  return promises;
}
```

```
var promises = doParallel ([add, sub], [[2,3],[4,5]]);
```

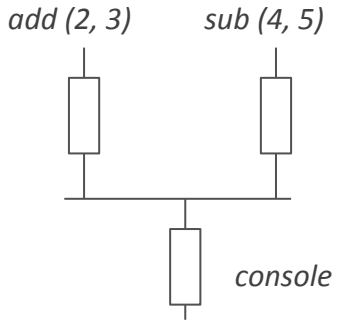
II. Control de Flujo Mediante Promesas

C. Sincronización

Dado que disponemos de los resultados potenciales en forma de promesas es fácil articular sobre ellos políticas de sincronización. El método **all** genera una promesa que toma una array de promesas y se resuelve a un array de valores cuando todas las promesas se han resuelto. Si hay fallo se devuelve el de la primera promesa del array en fallo. **allSettled** por su parte resuelve cuando todas las promesas estan en settled. Finalmente, **spread** es la versión en array del método then.

```
var promises = doParallel ([add, sub], [[2,3],[4,5]]);
Q.all (promises).then (function (values) {
  console.log (values);
});
```

```
var promises = doParallel ([mul, div], [[2,3],[4,0]]);
Q.allSettled (promises).spread (function (vMul, vDiv) {
  if (vMul.state === "fulfilled") console.log (vMul.value);
  if (vDiv.state === "fulfilled") console.log (vDiv.value);
});
```



III. Ejemplo

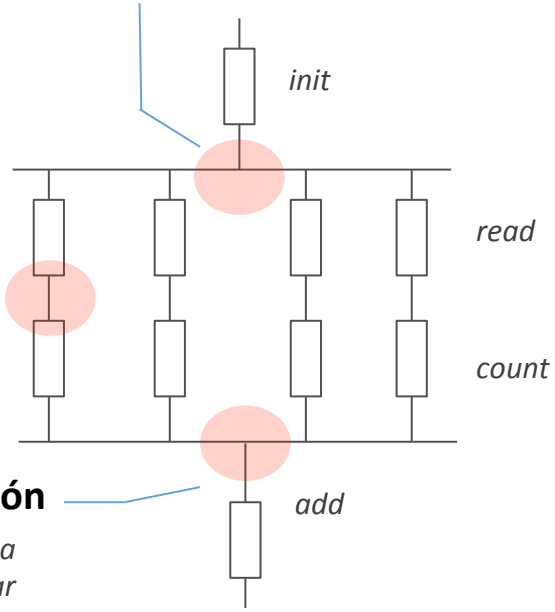
Promises.js

Dada una colección de ficheros, leer su contenido y contabilizar el número total de ocurrencias de cada carácter contenido dentro de los mismos

1. *Leer cada fichero en paralelo*
2. *Contabilizar su número de ocurrencias*
3. *Sumar los resultados*

Paralelización

Se lanzan a ejecución de forma iterativa todas las ramas paralelas y se construye un array de promesas para su posterior sincronización.



Secuenciamiento

El secuenciamiento consiste meramente en encadenar las funciones de lectura y conteo con dos métodos then.

Sincronización

Se recogen las promesas que resultan de cada rama secuencial en un array para poderlas sincronizar haciendo uso del método all.

IV. Librerías

Existen varias librerías que implementan el modelo de promesas. Tal vez la de mayor comunidad es **Q**, aunque otras como **When** o **RSVP**, también gozan de bastante tracción. En aras a disponer de un marco comparativo de referencia se ha definido el estándar **Promises A+** que rige todas todas las librerías con objetos then-able.

	Promises/A+	Progression	Delayed promise	Parallel synchronization
Bluebird	✓	✓ (+389 B)	✓ (+615 B)	✓ (+272 B)
Catiline	✓	-	-	✓
ES6 Promise polyfill	✓	-	-	✓
JQuery	-	✓	-	✓
kew	✓	-	✓	✓
lie	✓	-	-	-
MyDeferred	✓	-	-	✓
MyPromise	-	-	✓	✓
Q	✓	✓	✓	✓
RSVP	✓	-	-	✓
when	✓	✓	✓	✓
Yui	✓	✓	-	✓

V. Conclusiones

Lo Bueno

Hemos recuperado en gran parte el control de flujo del programa de manera que el esquema de desarrollo de aplicaciones asíncronas basadas en promesas se parece algo más al estilo secuencial manteniendo su carácter no bloqueante.

Recuperamos el return y la asignación

APIs más limpias sin métodos de callback (callback en cliente)

Estructura del programa más similar a la programación secuencial

Razonamos con promesas como valores de futuro

Lo Malo

Aunque este modelo es altamente prometedor y supone grandes ventajas con respecto al de paso de continuaciones, no deja de tener detractores que consideran el uso de promesas un artefacto demasiado artificial e incomodo de tratar.

No deja de ser necesario inyectar funciones manejadoras de éxito y error

Resulta difícil depurar hasta que las promesas no se han resuelto

Resulta más invasivo generar APIs que consuman y generen promesas

Es imprescindible hacer uso de librerías de terceros para gestionar las promesas

Javier Vélez Reyes
@javiervelezreye
Javier.veler.reyes@gmail.com

6

Modelo de Generadores

- Qué es un Generador
- Los Generadores como Modelo de Asincronía
- Control de Flujo Mediante Generadores
- Ejemplo
- Librerías
- Conclusiones

I. Qué es Un Generador

Generators.fib.js

Requiere

- *NodeJS > 0.11*
- *flag --harmony*

Imagina un procedimiento que pudiera ser interrumpido en su ejecución en cualquier punto antes de su terminación para devolver el control al programa llamante y más adelante éste volver a cederle el control para que continúe justo en el punto donde fue interrumpido y con el estado que tenía. Eso es un generador. Antes de ver su aplicación en modelos de asincronía veamos cómo funciona y cuál puede ser su aplicabilidad práctica.

Next

Inicia o reanuda la ejecución hasta el siguiente next

```
var fib = fibonacci() {
  for (var i=0; i<10; i++) {
    var f = fib.next();
    console.log(f.value);
  }
}

console.log (fib.next().value);
console.log (fib.next().value);
console.log (fib.next(true).value);
console.log (fib.next().value);
console.log (fib.next().value);
```

{ value: n, done: false }

```
function* fibonacci () {
  var a = 1;
  var b = 0;
  var aux;
  while (true){
    aux = a + b;
    a   = b;
    b   = aux;
    var reset = yield b;
    if (reset){
      a = 1;
      b = 0;
    }
  }
}
```

Yield

Suspende la ejecución y devuelve b

Reset

Obtiene como retorno el argumento de next

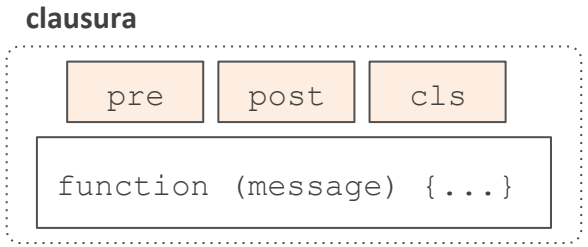
II. Los Generadores como Modelo de Asincronía

¿Qué es Una Clausura?

JS es un lenguaje de orden superior lo que implica que trata a las funciones como cualquier otro tipo de dato. Pueden asignarse a variables, pasarse como parámetros o devolverse como resultado. Lo bueno de esto último es que una función que es devuelta como resultado de invocar a otra función mantiene el contexto de variables definido dentro de esta última para su ejecución. Veamos un ejemplo:

```
function Logger(cls) {
  var pre  = 'Logger';
  var post = '...';
  return function (message) {
    console.log ('%s[%s] - [%s]%s',
                 pre, cls, message, post);
  }
}
```

```
var log = Logger ('Generators');
log('starting');
log(1234);
log('end');
```



Clausura

Al devolver una función hacia fuera del ámbito de definición se mantiene el contexto de variables y parámetros que dicha función tiene dentro, de forma trasparente. A ese contexto se le llama clausura

II. Los Generadores como Modelo de Asincronía

¿Qué es Evaluación Parcial de una Función?

En muchas ocasiones resulta conveniente convertir una función de n parámetros en otra que resuelve el primer parámetro y devuelve como resultado otra función con n-1 parámetros. A esto se le llama **evaluación parcial** de una función. Cuando la evaluación parcial se repite para cada parámetro de la función el proceso se llama **currificación**.

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

```
var r = add(3, 2);
```

Evaluación total

En la evaluación total el cliente debe esperar hasta obtener todos los parámetros actuales (3 y 2) para poder invocar a la función

```
function add(a) {
  return function (b) {
    return a + b;
  }
}
```

```
var r1  = add(3)(2);
var inc = add(1);
var r2  = inc(5);
```

Evaluación parcial

En la evaluación parcial cada parámetro (por orden) puede ser resuelto de manera independiente en sucesivas evaluaciones. Esto, como veremos, nos da la oportunidad de conseguir que sean distintos clientes los que realicen cada evaluación parcial

II. Los Generadores como Modelo de Asincronía

¿Qué es un Thunk?

En términos generales un **Thunk** es una abstracción computacional que sirve de vehículo para comunicar información entre dos frameworks diferentes. En nuestro caso queremos pasar de un modelo de invocación basado en continuaciones en otro que las oculte. Haciendo uso de los dos conceptos anteriores – clausuras y evaluación parcial – proponemos construir el siguiente thunk.

```
function add(a, b, callback){
  callback (null, x + y);
}
```

```
add(3, 2, function(error, data){
   console.log (data);
});
```

```
function add(a, b) {
  return function (callback) {
    callback (null, x + y);
  }
}
```

```
var r = add(3, 2);

r(function (error, data) {
   console.log (data);
});
```

Thunk

Nuestros thunks devuelven funciones que esperan manejadores como parámetros. De esta manera ocultamos el manejador en la primera evaluación parcial: add (2,3);

II. Los Generadores como Modelo de Asincronía

Generators.co.js

Requiere

- *NodeJS > 0.11*
- *flag --harmony*

Generadores y Thunks

Podemos construir un framework sobre el que escribir generadores que devuelvan thunks (con yield) lo que simplifica el código cliente. El framework por su parte va secuenciando el código del generador (con next) y se interpone para resolver de forma transparente la segunda evaluación parcial que corresponde con la función de callback. Su labor consiste en extraer los datos del callback y retornarlos al generador para que éste los obtenga como una variable local

Cliente

```
co (function* (){
  var r1 = yield add (2,3);
  var r2 = yield sub (3,4);
  console.log (r1, r2);
})();
```

Framework

```
var co = function (codeGn) {
  var code = codeGn ();
  function step(thunk) {
    if (thunk.done) return thunk.value;
    else {
      thunk.value (function (error, data){
        step(code.next (data));
      });
    }
  }
  return function () {
    step(code.next ());
  };
};
```

II. Control de Flujo Mediante Generadores

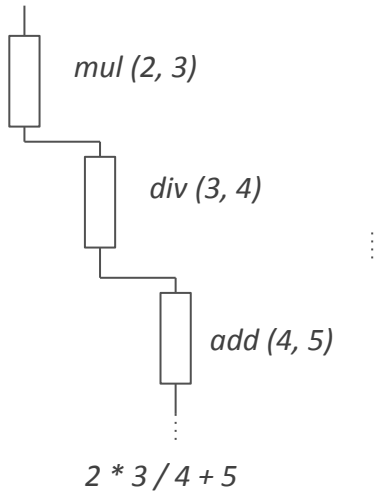
A. Secuenciamiento

De lo que hemos visto, el uso de yield dentro de un generador evaluado por nuestro framework asíncrono permite a éste interponerse para capturar la función de callback y resolver un thunk en su valor equivalente. De ello se deduce que cada vez que queramos secuenciar operaciones sólo tenemos que interponer yield antes de la operación.

```
co(function* (){

  var r1 = yield mul(2,3);
  var r2 = yield div(3,4);
  var r3 = yield add(4,5);

  console.log (r1, r2, r3);
})();
```



Yield en secuencia

Cada yield puede leerse como un paso dentro de una secuencia de operaciones. Cada operación no bloqueante (mul, div y add) devuelve un thunk que el framework, co, recoge para resolver a un valor y devolverlo al contexto del generador como una variable local (r1, r2 y r3)

II. Control de Flujo Mediante Generadores

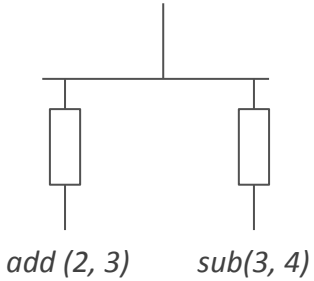
B. Paralelización & C. Sincronización

Razonablemente todas aquellas operaciones no bloqueantes que no sean intercedidas por el framework a través de la cláusula yield darán lugar a esquemas de ejecución paralelos. Es una práctica común agrupar estas invocaciones en colecciones (arrays u objetos) para posteriormente hacer la resolución de thunks a valores.

```
co(function* (){

  var r1 = add(2,3);
  var r2 = sub(3,4);
  var r  = yield [r1, r2];

  console.log (r);
})();
```



Yield de Array

Yield aplicado a un array se evalúa como el array de los yields. Yield aplicado a un objeto se evalúa como el yield aplicado a cada propiedad del objeto. De esta forma, este yield convierte el array de thunks en un array de valores

Array de thunks

Cada operación no bloqueante aquí es evaluada sin la clausula yield con lo que lo que se obtiene en r1 y r2 son dos thunks ejecutados en paralelo

III. Ejemplo

Generators.js

Requiere

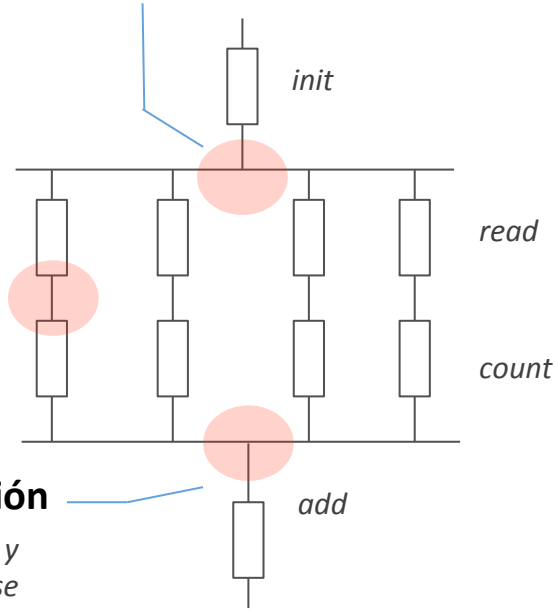
- *NodeJS > 0.11*
- *flag --harmony*

Dada una colección de ficheros, leer su contenido y contabilizar el número total de ocurrencias de cada carácter contenido dentro de los mismos

1. *Leer cada fichero en paralelo*
2. *Contabilizar su número de ocurrencias*
3. *Sumar los resultados*

Paralelización

Mediante un bucle, se construye un array de generadores para cada rama paralela



Secuenciamiento

Se construye un generador capaz de secuenciar las dos operaciones – read y count – que conforman cada rama paralela

Sincronización

Se hace un yield de los resultados para sincronizar y se invoca a add para obtener los totales, que se devuelven al cliente con otro yield ya que add también es no bloqueante

IV. Librerías

Aunque existen varias librerías que dan soporte a esta idea – articular un framework de soporte a la programación asíncrona basado en generadores – la que goza de mayor comunidad es **co**. En torno a ella se pueden encontrar una amplia colección de librerías utilitarias vinculadas que conforman el ecosistema de co.

Co

El framework de programación asíncrona basado en generadores con mayor aceptación. Co no sólo es capaz de operar con Thunks como abstracción vehicular sino que también funciona con promesas, funciones, generadores y funciones generadoras.

Thunkify

Se trata de una simple función que convierte a un thunk cualquier función definida de acuerdo al modelo de paso de continuidades estandarizado por Node JS.

Ecosistema Co*

Librerías específicas, adaptadores de APIs, servidores, funciones utilitarias de control de flujo… Todo un ecosistema de herramientas para programar dentro del framework co. Se puede ver una relación exhaustiva en la wiki de Co accesible desde github

V. Conclusiones

Lo Bueno

La programación asíncrona basada en generadores es, tal vez, la opción que mejor ha conseguido acercar la experiencia de desarrollo a la programación secuencial. A la fecha de esta charla aún es pronto para saber si será ampliamente aceptada por la comunidad o no.

Esquema de programación similar al secuencial

Transparencia en los procesos de gestión de continuaciones

Curva de aprendizaje corta. Con pocas reglas de pulgar se puede razonar fácilmente en el modelo

Lo Malo

El modelo de programación asíncrona basada en generadores sigue teniendo inconvenientes serios relacionados con la invasividad del framework y el uso permanente de clausulas yield para articular el manejo del control flujo.

Artificialidad del código. Perversión del uso de los yields

Todo código esta siempre dentro de un contexto Co

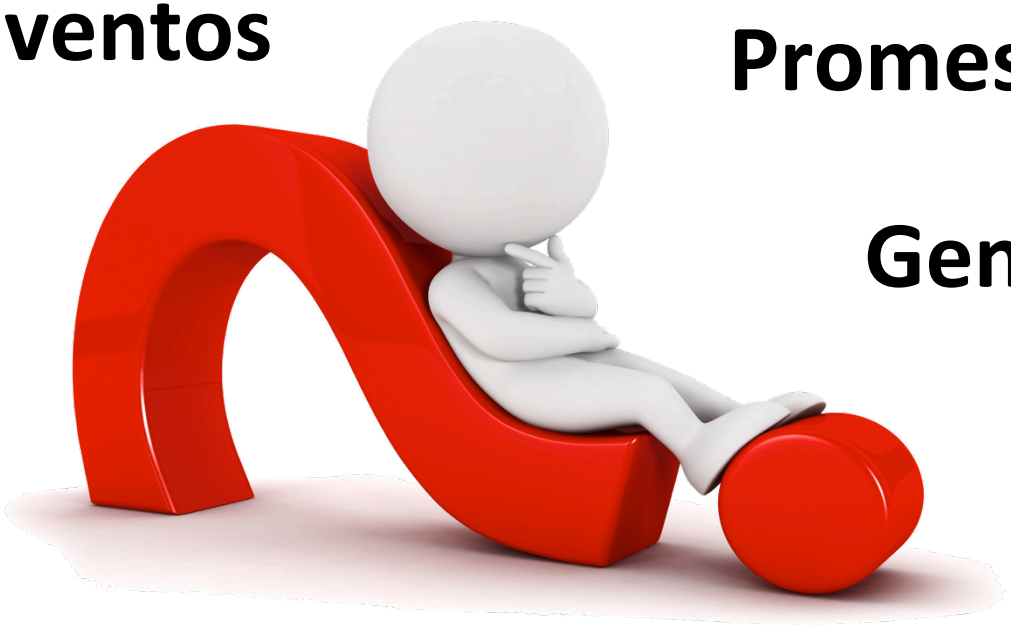
El código está plagado de yields

¿qué está haciendo Co por debajo?

Continuaciones

Eventos

Promesas



Generadores

Javier Vélez Reyes

@javiervelezreye
Javier.velez.reyes@gmail.com

Programación Asíncrona en Node JS

Modelo de Paso de Continuaciones, Eventos, Promesas y Generadores

Javier Vélez Reyes

@javiervelezreye
Javier.velez.reyes@gmail.com

Mayo 2014