ETSIT UPM

dit UPM

Promesas

CORE 2017-2018

Santiago Pavón

ver: 2018-02-26

Callbacks

- En JavaScript puede escribirse el código usando funciones síncronas.
  - Hasta que la función síncrona no ha terminado su tarea, no se ejecuta la siguiente sentencia.
  - El Main Thread está parado.
  - Poco recomendable para desarrollar servidores que tienen que atender muchas peticiones.

```
var fs = require('fs');

function lee(filename) {
    console.log(3);
    var datos = fs.readFileSync(filename, 'utf8');
    console.log(4);
    var res = JSON.parse(datos);
    console.log(5);
    return res;
}

console.log(1);
console.log(lee("datos.json"));
console.log(2);

// 1 3 4 5 DATOS 2
```

- Normalmente se usan funciones asíncronas para que el Main Thread no se quede parado.
  - En las llamadas a funciones asíncronas, no se detiene la ejecución del programa.
    - Se pasa a ejecutar la siguiente sentencia sin esperar a que la función asíncrona termine.
    - Y se siguen atendiendo los eventos que se producen.
  - Las funciones asíncronas toman como parámetro una (o varias) callback, que es la función a ejecutar cuando la función asíncrona ha terminado su tarea.

```
var fs = require('fs');

console.log(1);
fs.readFile("datos.json", 'utf8',
    function(err, datos) {
       console.log(3);
       if (err) {
           console.log("Error: ", err);
       } else {
           var res = JSON.parse(datos);
           console.log("Datos: ", res);
       }
       console.log(4);
    }
);
console.log(2);

// 1 2 3 Datos:??? 4
```

Problemas con Callbacks

- El código de los programas que usan callbacks puede ser poco claro.
  - El nivel de indentación crece mucho cuando unas callbacks están dentro de otras.

```
var fs = require('fs');
function copy(source, destination, callback) {
   fs.readFile(source, 'utf8', function(err, datos) {
      if (err) return callback(err);
      fs.writeFile(destination, datos, 'utf8', function(err) {
         if (err) return callback(err);
         callback(null);
      });
   });
}

copy("datos.json", "otro.json", function(err) {
   if (err) { console.log(err);
   } else { console.log("FIN"); }
});
```

- El código para ejecutar varias tareas asíncronas en paralelo también es complicado.

```
var res1, res2, res3;

tarea(datos1, function (result) {
    res1 = result;
    tareaTerminada();
});

tarea(datos2, function (result) {
    res2 = result;
    tareaTerminada();
});

tarea(datos3, function (result) {
    res3 = result;
    tareaTerminada();
});

var contadorTerminados = 0;

function tareaTerminada() {
    if (++contadorTerminados === 3) {
       console.log(res1, res2, res3);
    }
}
```

Resultados de las tareas.

Me invento variables y funciones para saber cuándo han terminado todas las tareas

- La gestión de los errores de las funciones asíncronas y de las callback está todo junto, y mezclado con el flujo normal de ejecución. El código no queda muy claro.

```
var fs = require('fs');

function lee(filename, callback) {
   fs.readFile(filename, 'utf8', function (err, datos) {
      if (err) return callback(err);
      var res;
      try {
         res = JSON.parse(datos);
      } catch(ex) {
         callback(ex);
         return;
      }
      callback(null, res);
 });
}
```

- El orden de ejecución puede variar:
  - Si el código que ejecuta la función **tarea** es síncrono se llama a la callback inmediatamente, y si asíncrono se llamará a la callback en el futuro.

```
console.log(1);

tarea("uno", function (result) {
      console.log(result);
});

console.log(2);

// 1 uno 2
```

```
console.log(1);

tarea("dos", function (result) {
      console.log(result);
});

console.log(2);

// 1 2 dos
```

```
function tarea(datos, callback) {
   if (datos == "uno") {
      callback(datos);
   } else {
      setTimeout(function() {
         callback(datos);
      }, 1000);
   }
}
```

En este ejemplo la llamada a la **callback** puede ser síncrona o asíncrona dependiendo del valor de la variable **datos**.

Promesas

- Para mejorar el código asíncrono se ha creado una abstracción llamada Promesas.
  - Permiten que el código los programas quede más sencillo.
  - El código de manejo de los errores y el comportamiento normal puede separarse.
    - Y se manejan de la misma manera.
  - Es muy fácil programar tareas en paralelo.
  - Su funcionamiento siempre es asíncrono.
  - . . .

¿Qué son las Promesas?

- Una Promesa es un objeto que representa el resultado de una tarea que se va a hacer en el futuro.
- A las promesas se las puede asociar manejadores para tratar los resultados que se obtengan en el futuro y los errores que se produzcan.
- Permiten programar métodos asíncronos como si fueran síncronos, simplemente devolviendo promesas de los valores buscados.
- Una promesa puede estar en tres estados:
  - Inicialmente, una promesa estará **pendiente** de que se ejecute la tarea que tiene asociada.
  - **Cumplida** cuando la tarea se haya ejecutado con éxito, y se obtuvo un resultado.
  - **Rechazada** cuando la tarea falla o se produce un error.

- La especificación de las promesas (Promises/A+) está disponible en:
  - https://promisesaplus.com

Construir una Promesa

- Creamos objetos llamando al constructor **Promise** pasando una función como argumento.
  - Esta función se ejecuta para obtener el resultado que debe devolver la promesa.
  - Esta función se ejecuta pasándola dos funciones como argumentos:
    - El primer argumento es la función a llamar para indicar que la promesa se ha cumplido con éxito.
      - Se la pasa como argumento el resultado que debe devolver la promesa.
    - El segundo argumento es la función a llamar para indicar que la promesa ha fallado.
      - Se la pasa como argumento un error que indique por qué falló la promesa.

```
new Promise(function (resolve, reject) {
   // Sentencias para hacer la tarea deseada.

   // Si todo va bien, la promesa se cumple satisfactoriamente
   // y llamamos a resolve con el resultado calculado.
    resolve(datos);

   // Si la promesa no se cumple, lo indicamos llamando a reject
   // pasando la razón del fallo.
   reject(new Error(msg));

   // Lanzar una excepción es equivalente que rechazar.
   throw new Error(msg);
});
```

```
var fs = require('fs');

new Promise(function(resolve, reject) {

    fs.readFile('file.txt', 'utf8', function (err, datos) {
        if (err) {
            reject(err);
        } else {
            resolve(datos);
        }
    });

}
```

Método asíncrono que en un futuro llamara a las funciones resolve o reject que nos han pasado en los argumentos.

Fallo: No he podido leer el fichero

He podido leer el fichero.
La promesa se cumple con éxito y su resultado son los datos leídos.

El método **then**

- Las promesas tienen un método **then** que permite estructurar el código de forma muy clara como un cadena de promesas.
  - Esto es así porque el método **then** devuelve otra promesa.
- El método **then** se usa para registrar en una promesa:
  - La función a ejecutar cuando la promesa se cumple.
    - Esta función toma como parámetro el valor calculado por la promesa.
  - La función a ejecutar cuando la promesa se rechaza.
    - Esta función toma como parámetro el valor con la razón por la que se rechazo la promesa.
    - Es opcional. Si no se indica, los rechazos se propagan devolviendo promesas con el mismo rechazo.
- La promesa devuelta por **then**:
  - se cumple si las funciones registradas devuelven un valor o una promesa que se cumpla, o no devuelven nada.
  - se rechaza si las funciones registradas lanzan una excepción o devuelven una promesa que se rechace.

```
var fs = require('fs');

var p1 = new Promise(function (resolve, reject) {
    fs.readFile('datos.json', 'utf8', function (err, datos) {
        if (err) {
            reject(err);
        } else {
            resolve(datos);
        }
    });
});

var p2 = p1.then(function(datos) {
                    return JSON.parse(datos);
                },
                function(err) {
                    return [];
                });

var p3 = p2.then(function(res) {
                    console.log(res);
                });
```

```
var fs = require('fs');

new Promise(function (resolve, reject) {
    fs.readFile('datos.json', 'utf8', function (err, datos) {
        if (err) {
            reject(err);
        } else {
            resolve(datos);
        }
    });
})
.then(function(datos) {
        return JSON.parse(datos);
      },
      function(err) {
        return [];
      }
)
.then(function(res) {
    console.log(res);
});
```

Mejor así

El método **catch**

- Las promesas tienen un método **catch** que también devuelve una promesa,
  - pero que solo atiende a las promesas rechazadas.
- Toma como parámetro la función a ejecutar cuando se ha rechazado la promesa.
  - Se le pasa como parámetro el valor con la causa del rechazo.
- Ejemplo:

```
promesa.catch(function(err) {
    console.log(err);
})
```

  - que es equivalente a:

```
promesa.then(undefined, function(err) {
     console.log(err);
})
```

```
new Promise(function (resolve, reject) {
    var d = [{name:'Pepe', age:22},
             {name:'Ana',  age:23},
             {name:'Luis', age:30}];
    resolve(d);
})
.then(function(people) {
    return people.map(function(p) {return p.age});
})
.then(function(ages) {
    return ages.reduce(function(a,b) { return a+b }, 0);
    })
.then(function(total) {
    console.log(total);   // 75
})
.catch(function(err) {
    console.log("Error:", err);
});
```

Nótese: Indentación mínima, código muy claro, manejo de errores aislado, …

Devolver una Promesa

- Hemos visto:
  - que cuando construimos una promesa, se llama a **resolve(valor)** para indicar que la promesa se cumplió satisfactoriamente y devuelve el valor **valor**, o se llama a **reject(error)** para indicar que falló y devuelve el error **error**.
  - que el método **then** devuelve una promesa que ejecuta una de las funciones que le pasan como parámetro, y que se cumple o falla según el valor que devuelto por la función ejecutada.
    - igual con **catch**.

```
new Promise(function(resolve, reject) {
    resolve(valor);  // Se crea una promesa que se resuelve satisfactoriamente
                     // devolviendo valor
})
.then(function(v) {
    return valor; // Esta promesa se satisface devolviendo valor
})
.then(function(v) {
    throw error; // Esta promesa falla devolviendo error
})
.catch(function(err) {
    return valor; // Esta promesa se satisface devolviendo valor
});
```

- Pero si **valor** es una promesa, entonces se espera a que se ejecute esta promesa, y según se ejecute con éxito o falle, se devuelve el valor o el error que genera.

```
new Promise(function(resolve, reject) {
    resolve("hola"); // Promesa se cumple y devuelve "hola"
})
.then(function(v) {
    // Espero a que la promesa se ejecute, y se devuelve lo que pase con ella.
    return new Promise(function(resolve, reject) {
        resolve(v.toUpperCase()); // Se ejecuta con exito y pasa v a mayusculas.
    });
})
.then(function(v) {
    console.log(v); // Imprime por consola HOLA
})
.catch(function(err) {
    console.log(err);
});
```

```
new Promise(function(resolve, reject) {
    resolve("hola"); // Promesa se cumple y devuelve "hola"
})
.then(function(v) {
    // No hay return. No se espera y se devuelve undefined inmediatamente.
    new Promise(function(resolve, reject) {
        resolve(v.toUpperCase());
    });
})
.then(function(v) {
    console.log(v); // Imprime por consola undefined
})
.catch(function(err) {
    console.log(err);
});
```

Los métodos **resolve** y **reject**

- **resolve** es un método de clase que devuelve una promesa que se resuelve con el valor que se pasa como argumento.
- **reject** es un método de clase que devuelve una promesa que se rechaza con el valor que se pasa como argumento.

```
var p = new Promise(function(resolve,reject) { resolve(1); });
Promise.resolve(p).then(function(v) { console.log(v); });
      // 1

Promise.resolve(2).then(function(v) { console.log(v); });
      // 2

Promise.reject(3)
   .then(function(v) { /* no se llama */ })
   .catch(function(err) { console.log(err); });
      // 3
```

El método **all**

- Método de clase que se usa para agregar varias promesas en una sola.
- Toma como argumento un objeto **iterable** (*String*, *Array*, *Map*, Set, …) conteniendo promesas o valores.
- Devuelve una promesa que:
  - se resuelve cuando todas las promesas del iterable se han resuelto.
    - el valor con el que se resuelve es un array con los valores con los que se resolvieron todas las promesas del iterable.
  - se rechaza si alguna de las promesas de iterable es rechazada.
    - Se rechaza con la misma razón que la promesa del iterable que fue rechazada.

```
var p1 = new Promise(function(resolve, reject) { resolve(1); });
var p2 = 22;
var p3 = new Promise(function(resolve, reject) { resolve(3); });

Promise.all([p1, p2, p3])
    .then(function(datos) { console.log(datos);  })
    .catch(function(err) { console.log(err); });
  // [ 1, 22, 3 ]
```

El método **race**

- Método de clase que toma como argumento un objeto **iterable** (*String, Array, Map,* Set, …) conteniendo promesas o valores.
- Devuelve una promesa que se resuelve o rechaza cuando una de las promesas del iterable se resuelve o rechaza,
  - y con el mismo resultado o valor de rechazo que la promesa que se resolvió o rechazó.

```
var p1 = new Promise(function(resolve, reject) { resolve(1); });
var p2 = 22;
var p3 = new Promise(function(resolve, reject) { resolve(3); });

Promise.race([p1, p2, p3])
    .then(function(dato) {
        console.log(dato);
    })
    .catch(function(err) {
        console.log(err);
    });
  // 1
```

Las Promesas son Asíncronas

- La ejecución de las promesas siempre es asíncrona.
  - Independientemente de que la operación que ejecuten sea síncrona o asíncrona.

```
new Promise(function(resolve, reject) {
    resolve(1);
})
.then(function(v) { console.log(v); });

new Promise(function(resolve, reject) {
    setTimeout(resolve,5000,2);
})
.then(function(v) { console.log(v); });

console.log("FIN")

// FIN  1  …5 segundos…  2
```

Implementaciones

- El soporte e implementaciones de Promesas es distinto en cada versión de JavaScript, de los navegadores e interpretes usados.
- Nodejs 4 soporta las características explicadas en las transparencias anteriores.
    - `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`
    - `https://nodejs.org/en/docs/es6`
- Una implementación más avanzada es la ofrecida por el paquete Bluebird.
    - `https://github.com/petkaantonov/bluebird`
    - `http://bluebirdjs.com`

Bluebird - API

- **Core**
  - new Promise
  - .then
  - .spread
  - .catch
  - .error
  - .finally
  - .bind
  - Promise.join
  - Promise.try
  - Promise.method
  - Promise.resolve
  - Promise.reject
- **Synchronous inspection**
  - PromiseInspection
  - .isFulfilled
  - .isRejected
  - .isPending
  - .isCancelled
  - .value
  - .reason
- **Collections**
  - Promise.all
  - Promise.props
  - Promise.any
  - Promise.some
  - Promise.reduce
  - Promise.filter
  - Promise.each
  - Promise.mapSeries
  - Promise.race
  - .all
  - .props
  - .any
  - .some
  - .map
  - .reduce
  - .filter
  - .each
  - .mapSeries
- **Resource management**
  - Promise.using
  - .disposer
- **Promisification**
  - Promise.promisify
  - Promise.promisifyAll
  - Promise.fromCallback
  - .asCallback
- **Timers**
  - .delay
  - .timeout
- **Cancellation**
  - .cancel
- **Generators**
  - Promise.coroutine
  - Promise.coroutine.addYieldHandler
- **Utility**
  - .tap
  - .call
  - .get
  - .return
  - .throw
  - .catchReturn
  - .catchThrow
  - .reflect
  - Promise.noConflict
  - Promise.setScheduler
- **Built-in error types**
  - OperationalError
  - TimeoutError
  - CancellationError
  - AggregateError
- **Configuration**
  - Global rejection events
  - Local rejection events
  - Promise.config
  - .suppressUnhandledRejections
  - .done
- **Progression migration**
- **Deferred migration**
- **Environment variables**