



[...] netmind

WeKnowIT

FICOSA

Taller Node

© 2017, Netmind SL, Barcelona

PROGRAMACIÓN FUNCIONAL

1. Recap arquitectura apps Node.js
2. Microservices, sockets, Redis
3. Apps multiprocesos
4. Colas de mensajes y mensajería redis
5. Node.js for Embedded Systems
6. Profiling de rendimiento
7. Proyecto Multiproceso con Colas

1

Recap arquitectura apps Node.js

Las 5 reglas fundamentales de una estructura de proyecto de Node.js

1. Organiza los archivos por características, no funciones

```
// DON'T
.
├── controllers
│   ├── product.js
│   └── user.js
├── models
│   ├── product.js
│   └── user.js
└── views
    ├── product.hbs
    └── user.hbs
```

- para entender cómo funciona una página del producto, hay que abrir tres directorios diferentes, con mucho cambio de contexto,
- Se termina escribiendo path largos cuando requiere módulos: require ('../../controllers / user.js')

```
// DO
.
├── product
│   ├── index.js
│   ├── product.js
│   └── product.hbs
└── user
    ├── index.js
    ├── user.js
    └── user.hbs
```

Las 5 reglas fundamentales de una estructura de proyecto de Node.js

2. No poner lógica en archivos index.js

- Sólo para exportar funcionalidades, como:

```
var product = require('./product')

module.exports = {
  create: product.create
}
```

Las 5 reglas fundamentales de una estructura de proyecto de Node.js

3. Los archivos de test junto a la implementación

- Las pruebas no son sólo para comprobar si un módulo produce la salida esperada.
- También documentan los módulos
- Es más fácil entender si los archivos de prueba se colocan junto a la implementación.

- Coloca los archivos de prueba adicionales en una carpeta de prueba separada (test) para evitar confusiones.

```
.
├── test
│   └── setup.spec.js
└── product
    ├── index.js
    ├── product.js
    ├── product.spec.js
    └── product.hbs
└── user
    ├── index.js
    ├── user.js
    ├── user.spec.js
    └── user.hbs
```

Las 5 reglas fundamentales de una estructura de proyecto de Node.js

4. Usar un directorio de configuración (config)

- Para los archivos de configuración.

```
.  
├── config  
│   ├── index.js  
│   └── server.js  
└── product  
    ├── index.js  
    ├── product.js  
    ├── product.spec.js  
    └── product.hbs
```

Las 5 reglas fundamentales de una estructura de proyecto de Node.js

4. Pon tus secuencias de comandos npm largo en un directorio de scripts

- Crea un directorio independiente para scripts largos que sean invocados desde package.json

```
.  
├── scripts  
│   ├── syncDb.sh  
│   └── provision.sh  
└── product  
    ├── index.js  
    ├── product.js  
    ├── product.spec.js  
    └── product.hbs
```

Estructura para proyectos complejos

- Estos son los objetivos que buscamos:
 - Escribir una aplicación que sea fácil de escalar y mantener.
 - La configuración separada de la lógica de negocio.
 - Nuestra aplicación puede consistir en múltiples tipos de procesos.

Estructura para proyectos complejos

- La estructura permite tener una configuración global y partir las configuraciones individuales de los componentes y workers
- Asimismo tener distintos procesos definidos por los workers
- Los modelos pueden ser compartidos entre distintos procesos
- Las variables de entorno pueden ser accedidas usando el objeto process.env

```
.-- config
|   |-- components
|   |   |-- comp1.js
|   |   |-- comp2.js
|   |   |-- ...
|   |-- index.js
|   |-- web.js
|-- models
|   |-- comp1
|   |   |-- index.js
|   |   `-- comp1.js
|   |-- comp2
|   |   |-- index.js
|   |   `-- comp2.js
|-- scripts
|-- test
`-- web
    |-- middleware
    |   |-- index.js
    |   `-- midd.js
    |-- router
    |   |-- api
    |   |   |-- comp1
    |   |   |   |-- module1.js
    |   |   |   |-- module1.spec.js
    |   |   |   `-- index.js
    |   |   `-- index.js
    |   `-- index.js
    |-- index.js
    `-- server.js
|-- worker
    |-- worker1
    |   |-- index.js
    |   `-- worker.js
    |-- worker2
    |   |-- index.js
    |   `-- worker.js
`-- index.js
`-- package.json
```



Analizando una aplicación multiproceso

➤ Clona la siguiente aplicación

- <https://github.com/RisingStack/multi-process-nodejs-example>
- by [András Tóth\(@tthndrs\)](#), Full-Stack Developer at RisingStack.

2

Microservices, sockets, Redis

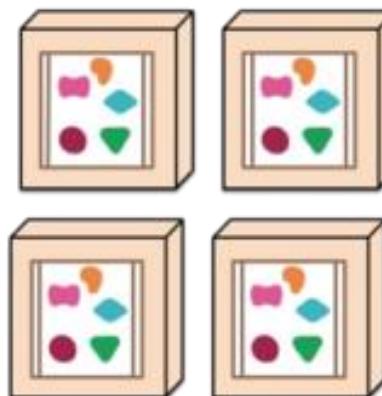
Microservicios

- Un microservicio es una sola unidad autónoma que, junto con muchos otros, constituye una gran aplicación. Al dividir tu aplicación en pequeñas unidades cada parte de ella es independientemente desplegable y escalable, puede ser escrito por diferentes equipos y en diferentes lenguajes de programación y puede ser probado individualmente. - **Max Stoiber**
- Una arquitectura de microservicios significa que tu aplicación está formada por un montón de aplicaciones más pequeñas e independientes que pueden ejecutarse en su propio espacio de memoria y escalar de forma independiente entre sí a través de potencialmente muchas máquinas separadas. - **Eric Elliot**

A monolithic application puts all its functionality into a single process...



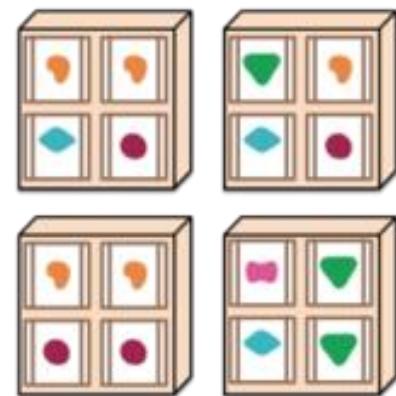
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



BENEFICIOS

- La aplicación se inicia más rápido, lo que hace que los desarrolladores sean más productivos y acelere las implementaciones.
- Cada servicio se puede implementar independientemente de otros servicios - más fácil de implementar nuevas versiones de servicios con frecuencia
- Más fácil de desarrollar a escala y también puede tener ventajas de rendimiento.
- Elimina cualquier compromiso a largo plazo con una pila tecnológica. Al desarrollar un nuevo servicio puede elegir una nueva pila de tecnología.
- Los microservicios suelen estar mejor organizados, ya que cada microservicio tiene un trabajo muy específico, y no se ocupa de los trabajos de otros componentes.
- Los servicios desacoplados también son más fáciles de recomponer y reconfigurar para servir a los propósitos de diferentes aplicaciones (por ejemplo, al servicio tanto de los clientes web como de la API pública).

INCONVENIENTES

- Los desarrolladores deben lidiar con la complejidad adicional de crear un sistema distribuido.
- Complejidad de implementación. En producción también existe la complejidad operacional de desplegar y administrar un sistema compuesto de muchos tipos de servicio diferentes.
- Como estás construyendo una nueva arquitectura de microservicios, es probable que descubra muchas preocupaciones transversales que no se anticiparon en el momento del diseño.

Construyendo microservicios con Node

- Como NodeJS está pensando para responder a eventos, es ideal para utilizar en esta arquitectura.
- Existen múltiples Frameworks, ya maduros como [Express](#), [Hapi](#), entre otros.
- [Hapi](#) está creado específicamente para desarrollar servicios [REST](#) de manera muy rápida y flexible.
- También [Hapi](#) nos permite crear las pruebas automáticas, usando un Framework de pruebas específico: [Lab](#).



Analizando un microservicio

➤ Clona el siguiente repositorio:

- <https://github.com/zankuda/hapi-ejemplo-1>
- Por Harold Fritz



Modificando una estructura de microservicio

- Modifica el proyecto movie-service-to-config para que tenga una estructura más correcta
- Necesitarás
 - MongoDB
 - Crear un usuario
 - use admin
 - db.createUser(

```
{  
    user: "root",  
    pwd: "xxx",  
    roles: [ "root" ]  
})
```
 - Cargar los datos en db movies y collection movies



Creando un microservicio con express

- Crea un microservicio que exponga las funcionalidades entorno a usuarios (get sobre users/:uid y post sobre users):
 - uid
 - nombre
 - apellido
 - email

Web Sockets

- Protocolo de comunicación (<https://tools.ietf.org/html/rfc6455>)
 - Full-duplex
 - Una sola conexión permanente
 - Stream de mensajes
 - Contenido en tiempo real
 - Orientado a “eventos” (mensajes)
 - Siempre conectado
 - Baja latencia
- Funcionan especialmente bien con Node.js
 - El servidor maneja muchas conexiones simultáneas
 - Buena integración con JSON
 - Eventos

Socket.io

- Una librería para manipular websockets
- Nos ofrece a la vez la parte servidor y la parte cliente
- Muy popular
- Fallback para navegadores obsoletos
- Muy fácil de usar
- Sin fricciones con express

- <http://socket.io/>

Socket.io – Servidor/cliente

Socket.io tiene dos partes:

- Servidor (Node.js):

```
var express = require("express"),
server = require("http").createServer(),
io = require("socket.io").listen(server),
app = express();
server.on("request", app).listen(3000);
```

- Cliente:

```
<script src="/socket.io/socket.io.js"></script>
```

Socket.io - Eventos

- Los sockets emiten eventos
- Un evento = un "mensaje"
- Se pueden pasar parámetros:
 - `socket.on(mensaje, callback)`
 - `socket.emit(mensaje, [param1, param2, ...])`
- Servidor:
 - `io.sockets.on("connection", cb)`
 - `socket.on("message", cb)`
 - `socket.on("disconnect", cb)`
- Cliente:
 - `socket.on("connect", cb)`
 - `socket.on("disconnect", cb)`
 - `socket.on("error", cb)`
 - `socket.on("message", cb)`

Socket.io - Métodos

➤ Métodos (servidor)

- socket.broadcast.emit(msg): les llega a todos menos el emisor
- socket.disconnect()
- socket.emit(msg) / socket.on(msg)

➤ Métodos (client)

- var socket = io.connect(host)
- socket.disconnect()
- socket.emit(msg) / socket.on(msg)

Socket.io - Callbacks

- El servidor puede llamar a callbacks en el cliente, a modo de respuesta a un mensaje

```
socket.emit("givemesomething", 5, function(obj){  
    console.log('Server answered', obj);  
});
```

- Servidor

```
socket.on('givemesomething', function(n, cb){  
    cb(n*10);  
})
```

Socket.io – Canales

- Podemos tener varios canales simultáneos
- Multiplexando el mismo websocket

```
io.of("/canal").on("connection", function(socket) {  
    socket.emit("ping");  
});  
  
io.of("/otro").on("connection", function(socket) {  
    socket.emit("bang!");  
});
```

- En cliente:

```
var socket = io.connect("http://localhost:3000/canal");
```

Socket.io – Salas

- Además de namespaces, socket.io incluye el concepto de sala
- Sólo el servidor mete y saca sockets de una sala
- `socket.join("room") / socket.leave("room")`
- Además se puede enviar (o hacer broadcast) a una sala específica:

```
io.to("room").emit(...)  
io.to("room").broadcast.emit(...)
```

APIs sobre WebSockets

- Si estas operaciones se realizan mediante WebSocket, tenemos una oportunidad para notificar al resto de usuarios conectados de que se ha creado/modificado/borrado un nuevo algo
- Evitamos el problema de que cada usuario vea una copia "congelada" de los datos en el servidor

```
// Versión REST
```

```
GET /posts - devuelve la lista de Posts (SELECT)
POST /posts - crea un nuevo Post (INSERT)
GET /posts/id - devuelve un Post (SELECT 1)
PUT /posts/id - modifica un Post (UPDATE)
DELETE /posts/id - elimina un Post (DELETE)
```

```
// Versión Socket
```

```
socket.emit("posts:list", function(posts){...})
socket.emit("posts:create", postObj, function(post){...})
socket.emit("posts:get", postId, function(post){ ... })
socket.emit("posts:update", postObj, function(post){...})
socket.emit("posts:delete", function(result){...})
//Aviso: nuevo Post!
socket.on("posts:create", function(post){...})
```

APIs – REST vs Websocket

```
// Versión REST
```

GET /posts - devuelve la lista de Posts (SELECT)
POST /posts - crea un nuevo Post (INSERT)
GET /posts/id - devuelve un Post (SELECT 1)
PUT /posts/id - modifica un Post (UPDATE)
DELETE /posts/id - elimina un Post (DELETE)

```
// Versión Socket
```

```
socket.emit("posts:list", function(posts){...})  
socket.emit("posts:create", postObj, function(post){...})  
socket.emit("posts:get", postId, function(post){ ... })  
socket.emit("posts:update", postObj, function(post){...})  
socket.emit("posts:delete", function(result){...})  
//Aviso: nuevo Post!  
socket.on("posts:create", function(post){...})
```



Una API con Web Sockets

- Crea una API basada en sockets que exponga las funcionalidades de usuarios

Redis

- Una base de datos NoSQL...
 - Clave/valor
 - Extremadamente rápida
 - Persistencia y transacciones
 - Estructuras de datos
 - - Listas
 - - Conjuntos
 - - Hashes
 - Pub/sub
- Un servidor de estructuras de datos
 - Sin tablas, ni queries ni JOINs
 - Una manera de pensar muy diferente
 - Muchas queries muy rápidas
 - Muy fácil de aprender, muy fácil de usar mal

Redis – Casos de uso

- Redis es una opción estupenda para...
 - Datos de acceso inmediato
 - Sesiones
 - Cachés
 - Escrituras muy rápidas
 - Logs
 - Conteos y estadísticas
 - Canal de comunicación pub/sub
 - Comunicación entre procesos (workers, big data)
 - Chats
- Redis es una opción terrible para...
 - Relaciones de datos complejas
 - Búsquedas

Redis – Hash de claves

- No hay concepto de tabla/colección
- El valor fundamental es la cadena (string)
- Pero una clave puede contener un valor más complejo:
 - Hashes (string -> string)
 - Listas (de strings)
 - Sets (de strings)
 - Sets ordenados (de strings)



Instalar redis

Redis – Set/Get

➤ Conexión, set, get

```
var redis = require("redis"),
client = redis.createClient();

client.set("miClave", "miValor", function(err, val) {
    console.log(arguments);
    client.get("miClave", function(err, value) {
        console.log("valor: ", value);
    });
})
```

- Usar el monitor:
 - redis-cli monitor

Redis – DEL / EXISTS / TYPE / RENAME

- DEL: borrar una clave
- EXISTS: comprobar si una clave existe
- TYPE: el tipo de valor almacenado en una clave
- RENAME: cambiar el nombre de la clave

Redis – Operaciones con cadenas y múltiples

- APPEND: añade el valor a la cadena
 - DECR/INCR: Decrementa/incrementa el valor en 1
 - DECRBY/INCRBY: Dec/inc el valor en N
 - GETSET: Modifica el valor y devuelve el viejo
 - STRLEN: Longitud del valor
-
- MGET: Trae el valor de varias claves
 - MSET: Modifica el valor de varias claves

Redis – Listas

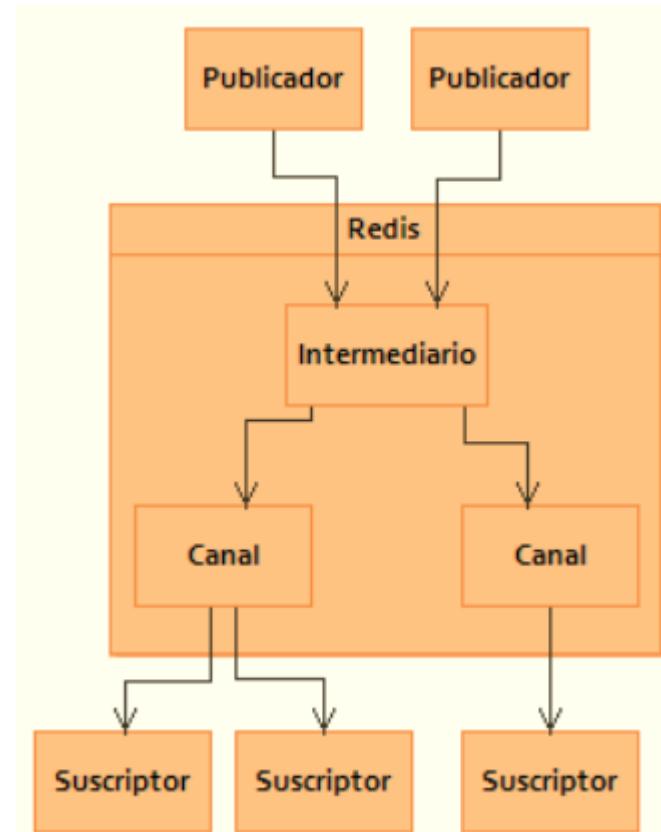
- LPUSH/RPUSH key value [value ...]
- LPOP/RPOP key
- LINDEX key index
- LSET key index value
- LLEN key
- LRANGE key start stop: trae el rango de elementos - si stop es -1 trae todos
- LTRIM key start stop: recorta al rango start-stop
- RPOPLPUSH source dest: RPOP sour + LPUSH dest

Redis – Hashes

- HSET key field value: Modifica el valor de campo field del hash en value
- HGET key field: Consulta el valor de campo field del hash en value
- HEXISTS key field: Existe el campo field?
- HKEYS/HVALS key: Todos los campos/valores
- HGETALL: Trae el hash entero
- HINCRBY key field n: Incrementa el campo en n
- HMGET/HMSET: Operaciones múltiples

Redis – Mensajería

- Una característica muy valorada de Redis es su servicio de mensajería integrado que integra el modelo publicación/suscripción (publish/subscribe).
- Un canal (channel) o tema (topic) es un contenedor donde se publican mensajes. Es el medio a través del cual los extremos intercambian mensajes.
- El intermediario es el responsable de administrarlos. Una vez los ha entregado, los suprime automáticamente.
- Los emisores de los mensajes se los envían al intermediario indicándole a través de qué canales debe difundirlos (publication).
- Mientras que cuando un componente desea recibir mensajes publicados por otros lo que hace es abonarse a los canales que son de su interés (subscription).
- Cada vez que el intermediario publica un mensaje en un canal, cuando pueda se lo enviará a sus suscriptores y, finalizado, lo suprimirá del canal.



Redis – Mensajería con CLI

- Usando el cli de redis podemos publicar y suscribirnos a canales.
- Publicar
 - redis-cli
 - > PUBLISH channel message
 - > PUBLISH alta "{'correo': 'elvis@costello.com', 'nombre': 'Elvis'}"
- Suscribirse/De-suscribirse (se puede suscribir a más de uno)
 - redis-cli
 - > SUBSCRIBE canal canal canal...
 - > UNSUBSCRIBE canal canal canal...
 - > SUBSCRIBE alta



Microservicio con Redis

- Modifica tu microservicio para usar Redis

Redis – PUB/SUB

- El comando PUBSUB se utiliza para obtener información del servicio de mensajería: los canales existentes y los suscriptores.
- Para conocer los canales, se utiliza la siguiente sintaxis:
 - PUBSUB CHANNELS
 - PUBSUB CHANNELS [argument]
 - Si no se indica ningún argumento, mostrará todos los existentes actualmente.
 - Si se indica un patrón, aquellos cuyo nombre coincide con el indicado.
- Para conocer el número de suscriptores de uno o más canales indicados, se utiliza:
 - PUBSUB NUMSUB canal canal canal...
 - redis-cli
 - > PUBSUB CHANNELS
 - > PUBSUB NUMSUB alta baja

Integración con Node

- Redis provee dos métodos para suscribirse y publicar en canales de mensajería (<https://www.npmjs.com/package/redis>)
 - publish(channel, cb)
 - subscribe(channel)
 - Eventos:
 - message
 - subscribe
 - unsubscribe
 - unsubscribe()
- Hay que tener en cuenta que un cliente cuando publica ya no puede tener otro rol, por tanto será necesario crear dos clientes para una comunicación completa

Ejemplo

```
var redis = require("redis");
var sub = redis.createClient(), pub = redis.createClient();
var msg_count = 0;

sub.on("subscribe", function (channel, count) {
    pub.publish("a nice channel", "I am sending a message.");
    pub.publish("a nice channel", "I am sending a second
message.");
    pub.publish("a nice channel", "I am sending my last
message.");
});

sub.on("message", function (channel, message) {
    console.log("sub channel " + channel + ": " + message);
    msg_count += 1;
    if (msg_count === 3) {
        sub.unsubscribe();
        sub.quit();
        pub.quit();
    }
});

sub.subscribe("a nice channel");
```



Cola de mensajes con Redis

- Modifica tu microservicio para sincronizar su estado contra Redis
- Crea un cliente que permita publicar/recibir los estados

3

Apps multiprocesos

Porque crear procesos hijos en Node

- Cuando se realiza un uso intensivo de la CPU o un gran número de tareas, puede que el rendimiento se vea perjudicado y tener un aumento en el tiempo de ejecución / respuesta.
- Esto puede tener un impacto fuerte en aplicaciones web ya que Node se puede volver bloqueante.
- Si una aplicación NodeJS tiene un 100% de uso de la CPU y tarda mucho tiempo en completarse o tardar en responder, se puede mejorar dividiendo el trabajo que se va a realizar y extendiéndolo en varios procesos.

Multiproceso en Node

- Una aplicación Node.js se ejecuta en un solo hilo. En este hilo, un bucle [de eventos] escucha los eventos y, a continuación, activa la función de callback asociada tras su detección.
- Con esta sencilla ilustración ya podemos ver que Node.js no admite multithreading. Aunque el multithreading no es compatible, hay una manera de aprovechar la potencia de un sistema multinúcleo mediante el uso de procesos.
- Como desarrollador de Node.js, si deseas escalar el servidor Node.js para que se ejecute en varios núcleos de CPU, tendrá que implementarlo como varios procesos.
- Node.js ofrece dos excelentes módulos básicos para tratar múltiples procesos:
 - child_process: http://nodejs.org/api/child_process.html
 - clúster: <http://nodejs.org/api/cluster.html>

Ejemplo con cluster

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  // Fork workers.
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(8000);
}
```

Threads vs Procesos

- Hemos logrado implementar con éxito una aplicación Node.js multinúcleo sin embargo, con cada decisión de diseño hay trade-offs a evaluar.
- En este caso nuestros trade-offs están asociados con los conceptos de multithreading y multiprocesamiento.
- Los threads comparten el mismo espacio de memoria y la creación de nuevos hilos no ocupa gran parte de los recursos del sistema.
- Los procesos se ejecutan en asignaciones de memoria independientes con una copia completa del programa que aumenta la sobrecarga de memoria de la aplicación.
- En situaciones donde la memoria podría ser escasa, Node.js podría ser una herramienta poco eficaz para el código de aplicación.

child_process

- Permite generar un proceso secundario y esos procesos secundarios pueden comunicarse fácilmente entre sí con un sistema de mensajería.
- El módulo child_process nos permite acceder a las funcionalidades del Sistema Operativo ejecutando cualquier comando del sistema.
- Podemos controlar ese flujo de entrada de proceso hijo y escuchar su flujo de salida. También podemos controlar los argumentos que se van a pasar al comando OS subyacente, y podemos hacer lo que queramos con la salida de ese comando.
 - Podemos, por ejemplo, canalizar la salida de un comando como entrada a otro (como lo hacemos en Linux) ya que todas las entradas y salidas de estos comandos pueden ser presentadas usando Node.js streams.
- Existen cuatro maneras diferentes de crear un proceso secundario en Nodo: spawn(), fork(), exec() y execFile().

child_process: spawn()

- Spawn se puede utilizar para crear un proceso desde cualquier comando.

```
const spawn = require('child_process').spawn;

const command = 'node';
const parameters = [path.resolve('program.js')];

const child = spawn(command, parameters, {
  stdio: [ 'pipe', 'pipe', 'pipe', 'ipc' ]
});
```

child_process: fork()

- Es una variación de la función spawn para generar subprocessos.
- La diferencia más importante entre spawn y fork es que fork establece un canal de comunicación al proceso hijo
- Por lo que podemos usar la función send en el proceso bifurcado junto con el objeto global del proceso para intercambiar mensajes entre padre e hijos.
- Lo hacemos a través de la interfaz del módulo EventEmitter.
- Los procesos hijos normalmente se implementan en un script aparte.
- A estos scripts secundarios se les conoce como workers

Ejemplo

```
// parent.js

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

const forked = fork('child.js');

forked.on('message', (msg) => {
  console.log('Message from child', msg);
});

forked.send({ hello: 'world' });
```

```
// child.js

process.on('message', (msg) => {
  console.log('Message from parent:', msg);
});

let counter = 0;

setInterval(() => {
  process.send({ counter: counter++ });
}, 1000);
```

Comunicación entre procesos

- Para comunicarse con el proceso hijo necesitamos habilitar el ipc(inter process communication).
- Para hacer esto necesitamos agregar una lista de opciones de entrada / salida.

```
const fork = require('child_process').fork;

const program = path.resolve('program.js');
const parameters = [];
const options = {
  stdio: [ 'pipe', 'pipe', 'pipe', 'ipc' ]
};

const child = fork(program, parameters, options);
```

- Esto habilitará los métodos `process.send()` y `process.on('mensaje')` en el proceso secundario; para comunicarse entre los 2 procesos.

Comunicación entre procesos

```
// child.js

if (process.send) {
  process.send("Hello");
}

process.on('message', message => {
  console.log('message from parent:', message);
});
```

```
// parent.js

const fork = require('child_process').fork;

const program = path.resolve('child.js');
const parameters = [];
const options = {
  stdio: [ 'pipe', 'pipe', 'pipe', 'ipc' ]
};

const child = fork(program, parameters, options);
child.on('message', message => {
  console.log('message from child:', message);
  child.send('Hi');
});
```

Comunicación entre procesos no relacionados

- Si hay dos procesos independientes y se quiere que se comuniquen, se puede hacer de forma fiable con el paquete: node-ipc (<https://www.npmjs.com/package/node-ipc>)
- Funciona mediante la creación de un sistema de pub-sub global, al que puede suscribirse y publicar desde cualquier proceso Node.

Comunicación entre procesos no relacionados

```
const ipc = require('node-ipc');

ipc.config.id = 'a-unique-process-name1';
ipc.config.retry = 1500;
ipc.config.silent = true;
ipc.serve(() => ipc.server.on('a-unique-message-
name', message => {
  console.log(message);
}));
ipc.server.start();
```

```
const ipc = require('node-ipc');

ipc.config.id = 'a-unique-process-name2';
ipc.config.retry = 1500;
ipc.config.silent = true;
ipc.connectTo('a-unique-process-name1', () => {
  ipc.of['jest-observer'].on('connect', () => {
    ipc.of['jest-observer'].emit('a-unique-message-
name', "The message we send");
  });
});
```

Hacer uso de todos los núcleos de la CPU para un solo proceso NodeJS: clustering

- child_process también se puede usar para el clúster: ejecutando varias instancias del mismo programa y equilibrando una gran carga de trabajo sobre estos procesos.
- En NodeJS 7 se añadió una nueva api: **cluster**.
- Permite que la creación de un servidor que utilice todos los núcleos sea sencillo.
- Cluster se ha construido sobre child_process

Ejemplo de cluster

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
    // Fork workers.
    for (var i = 0; i < numCPUs; i++) {
        cluster.fork();
    }

    cluster.on('exit', (worker, code, signal) => {
        console.log(`worker ${worker.process.pid} died`);
    });
} else {
    // Workers can share any TCP connection
    // In this case it is an HTTP server
    http.createServer((req, res) => {
        res.writeHead(200);
        res.end('hello world\n');
    }).listen(8000);
}
```

Cluster

- Los workers pueden comunicarse a través de ipc entre sí o con todo el clúster utilizando el evento de mensaje.

```
cons cluster = require('cluster');
const http = require('http');

if (cluster.isMaster) {
  // init cluster
  require('os').cpus().forEach(() => {
    cluster.fork();
  });

  // add eventlisteners
  Object.values(cluster.workers).forEach(worker => {
    worker.on('message', message => {
      console.log(message);
    });
  });
} else {

  http.Server((req, res) => {
    res.send(200, 'hello world\n');

    process.send('Hi');
  }).listen(8000);
}
```



Generando multiprocesos

- Transforma el siguiente programa en multiproceso para evitar el bloqueo del servidor

```
const http = require('http');
const longComputation = () => {
  let sum = 0;
  for (let i = 0; i < 1e9; i++) {
    sum += i;
  };
  return sum;
};
const server = http.createServer();
server.on('request', (req, res) => {
  if (req.url === '/compute') {
    const sum = longComputation();
    return res.end(`Sum is ${sum}`);
  } else {
    res.end('Ok')
  }
});

server.listen(3000);
```

4

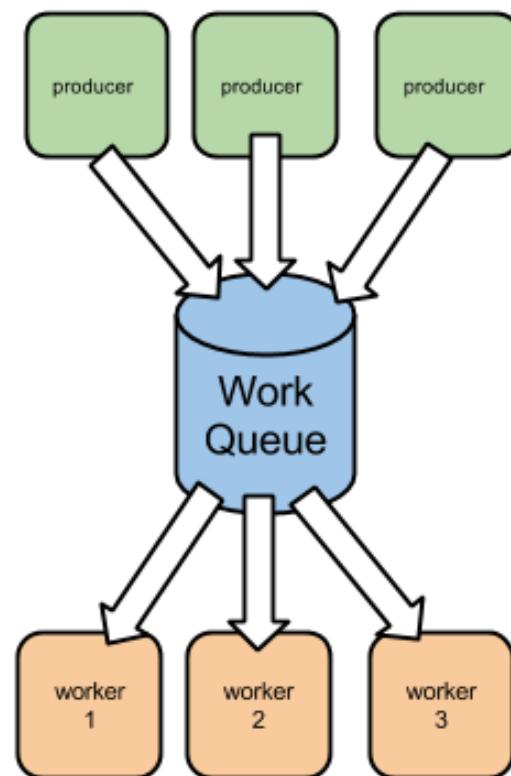
Colas de mensajes y mensajería redis

Colas en las aplicaciones

- Es común que las aplicaciones tengan cargas de trabajo que se puedan procesar de forma asincrónica desde los flujos de aplicaciones.
 - Un ejemplo común de esto es el envío de un correo electrónico. Por ejemplo, cuando un nuevo usuario se registra, se le envía un correo electrónico de confirmación para confirmar que la dirección. Esto implica componer el mensaje de una plantilla; hacer una solicitud a un proveedor de correo electrónico; analizar el resultado; manejar cualquier error eventual que pueda ocurrir; reintentar, etc.
 - Este flujo puede ser demasiado complejo, propenso a errores o tomar demasiado tiempo para incluirlo en el ciclo de un servidor HTTP.
- Además, es muy común que los sistemas tengan que integrarse con otros sistemas

Colas: Productores <-> Consumidores

- Más generalmente, es un patrón común a través de sistemas tener una cola del trabajo que separe los productores del trabajo de los consumidores del trabajo.
- Los productores insertar trabajo en una cola de trabajo y los consumidores pop trabajo de la cola, la realización de las tareas necesarias.



Colas: beneficios

Hay muchas razones y ventajas en el uso de tal topología, tales como:

- Desacoplamiento de productores y consumidores
- Hacer que la lógica de reintento sea más fácil de implementar
- Distribución de la carga de trabajo a lo largo del tiempo
- Distribución de la carga de trabajo en el espacio (nodos)
- Trabajo asincrónico
- Facilitar la integración de los sistemas externos (eventual coherencia)

Implementación en Node

- Hay muchas maneras de implementar colas en Node
- Una de ellas es utilizar la primitiva queue de Async (<https://github.com/caolan/async/blob/v1.5.2/README.md>) para crear una cola de trabajo (en memoria).
- Otra opción es usar co-queue (<https://www.npmjs.com/package/co-queue>) que crea una cola FIFO para co (<https://www.npmjs.com/package/co>)
 - Control de flujo basado en Generador de flujo para nodejs y el navegador, usa promesas.
- Otra librería para procesado de trabajos en cola es Kue (<https://github.com/Automattic/kue>). Se integra con redis.

Ejemplo: productor y dos consumidores

```
var Queue = require('co-queue');
var co = require('co');
var wait = require('co-wait');

var queue = new Queue;

co(function*(){
  while (true) {
    console.log('consumer 1: %s', yield queue.next());
    yield wait(Math.random() * 1000);
  }
});

co(function*(){
  while (true) {
    console.log('consumer 2: %s', yield queue.next());
    yield wait(Math.random() * 1000);
  }
});

setInterval(function(){
  queue.push(Math.random());
}, 300);
```

Un ejemplo más complejo: cola domótica



- Imaginemos que estamos construyendo esta aplicación domótica que interactúa con una unidad de hardware que controla la casa.
- Su aplicación Node.js habla con esta unidad utilizando un puerto serie y el protocolo de cable sólo acepta un comando pendiente a la vez.
- Este protocolo puede ser encapsulado dentro de este módulo domotic.js que exporta tres funciones:
 - .connect () - para conectarse al módulo domótico
 - .command () - para enviar un comando y esperar la respuesta
 - .disconnect () - para desconectarse del módulo

domotic.js

```
exports.connect = connect;
exports.command = command;
exports.disconnect = disconnect;

function connect(cb) {
    setTimeout(cb, 100); // simulate connection
}

function command(cmd, options, cb) {
    if (succeeds()) {
        setTimeout(cb, 100); // simulate command
    } else {
        setTimeout(function() {
            var err = Error('error connecting');
            err.code = 'ECONN';
            cb(err);
        }, 100);
    }
}

function disconnect(cb) {
    if (cb) setTimeout(cb, 100); // simulate disconnection
}

function succeeds() {
    return Math.random() > 0.5;
}
```



domotic_queue.js



```
var async = require('async');
var Backoff = require('backoff');
var domotic = require('./domotic');

var connected = false;

var queue = async.queue(work, 1);

function work(item, cb) {
  ensureConnected(function() {
    domotic.command(item.command, item.options, callback);
  });
}

function callback(err) {
  if (err && err.code == 'ECONN') {
    connected = false;
    work(item);
  } else cb(err);
}
}

/// command

exports.command = pushCommand;

function pushCommand(command, options, cb) {
  var work = {
    command: command,
    options: options
  };

  console.log('pushing command', work);
  queue.push(work, cb);
}

function ensureConnected(cb) {
  if (connected) {
    return cb();
  } else {
    var backoff = Backoff.fibonacci();
    backoff.on('backoff', connect);
    backoff.backoff();
  }
}

function connect() {
  domotic.connect(connected);
}

function connected(err) {
  if (err) {
    backoff.backoff();
  } else {
    connected = true;
    cb();
  }
}

/// disconnect

exports.disconnect = disconnect;

function disconnect() {
  if (!queue.length()) {
    domotic.disconnect();
  } else {
    console.log('waiting for the queue to drain before disconnecting');
    queue.drain = function() {
      console.log('disconnecting');
      domotic.disconnect();
    };
  }
}
```

client.js



- Creamos el cliente

```
var domotic = require('./domotic_queue');

for(var i = 0 ; i < 20; i++) {
    domotic.command('toggle light', i, function(err) {
        if (err) throw err;
        console.log('command finished');
    });
}

domotic.disconnect();
```

Colas con persistencia

- Si, durante un período de tiempo, la producción de mensajes excede la capacidad del worker, la cola puede comenzar a consumir demasiada memoria.
- Además, si el proceso de Node.js se cae, perderá cualquier trabajo pendiente que tuviera.
- Al trasladar la cola de trabajo fuera de memoria al almacenamiento persistente, podemos evitar estos problemas:
 - Al mantener sólo el trabajo actual en la memoria, podemos absorber los picos de trabajo, ya que se escribirá en el disco.
 - Además, cuando se inicia el proceso Node.js, se reanuda cualquier trabajo pendiente.

Cola con persistencia



- Supongamos que se quiere incorporar un sistema de alarma y se desea retransmitir cada evento (alarma apagada, alarma encendida, puerta abierta, alerta, etc.), en un servicio remoto para almacenamiento y futuro fines de la auditoría.
- Este servicio remoto puede estar inactivo o inaccesible a veces, pero todavía desea que los eventos finalmente se ejecuten.
- A continuación, construimos un módulo que crea y exporta una cola persistente donde se almacenarán los eventos de seguridad.
- Este módulo también será responsable de retransmitir estos eventos en el servicio remoto.
- Usaremos LevelDB (<http://leveldb.org/>) y el módulo level-Jobs (<https://www.npmjs.com/package/level-jobs>) para encolar trabajos

Almacén de eventos



- Construimos un módulo que crea y exporta una cola persistente donde se almacenarán los eventos de seguridad.
- Este módulo también será responsable de retransmitir estos eventos en el servicio remoto.

```
// event_relay.js

var level = require('level');
var db = level('./db');

var Jobs = require('level-jobs');

var maxConcurrency = 1;
var queue = Jobs(db, worker, maxConcurrency);

module.exports = queue;

function worker(event, cb) {
  sendEventToRemoteService(event, function(err) {
    if (err) console.error('Error processing event %s: %s', event.id, err.message);
    else console.log('event %s successfully relayed', event.id);
    cb(err);
  });
}

function sendEventToRemoteService(event, cb) {
  setTimeout(function() {
    var err;
    if (Math.random() > 0.5) err = Error('something awful has happened');
    cb(err);
  }, 100);
}
```

Productor de eventos



- Construimos un módulo que crea y exporta una cola persistente donde se almacenarán los eventos de seguridad.
- Este módulo también será responsable de retransmitir estos eventos en el servicio remoto.

```
// event_producer.js

var relay = require('./event_relay');

for(var i = 0 ; i < 10; i++) {
  relay.push({id: i, event: 'door opened', when: Date.now()});
}
```

Cola distribuida

- La creación de una cola local puede ser muy útil para agilizar la carga de trabajo y persistir el trabajo a lo largo de los reinicios del proceso.
- Sin embargo, dependiendo del tipo de trabajo que se esté haciendo, este enfoque podría presentar algunos problemas:
 - Si este trabajo es de alguna manera intensivo en consumo de CPU puede que se necesite externalizarlo a un conjunto de procesos externos, disociando aún más la producción del consumo de trabajo.
- Node.js y JavaScript en general no son particularmente adecuados para realizar cálculos de larga duración.
- Usaremos redis y el módulo para encolado simple-redis-safe-work-queue (<https://github.com/pgte/simple-redis-safe-work-queue>) que usa redis
- Otras opciones son:
 - Bull (<https://github.com/OptimalBits/bull>)
 - Kue (<https://github.com/Automattic/kue>)

worker.js

```
var request = require('request');
var Queue = require('simple-redis-safe-work-queue')

var worker = Queue.worker('invoke webhook', invokeWebhook);

function invokeWebhook(webhook, cb) {
  console.log('invoke webhook: %j', webhook);

  request(webhook, done);

  function done(err, res) {
    if (!err && (res.statusCode < 200 || res.statusCode >= 300)) {
      err = Error('response status code was ' + res.statusCode);
    }
    cb(err);
  }
}

worker.on('max retries', function(err, payload) {
  console.error(
    'max retries reached trying to talk to %s.: %s\nrequest params: %j',
    payload.url, err.stack, payload);
});
```

client.js

```
var Queue = require('simple-redis-safe-work-queue')

var queueClient = Queue.client('invoke webhook');

queueClient.push({
  url: 'http://example.com',
  method: 'POST',
  json: {
    a: 1,
    b: 2
  }
});

queueClient.stop();
```

client.js –boot/shutdown

- Si se está produciendo trabajo en el contexto de un servicio en ejecución, es mejor crear el cliente en el momento del arranque (boot) y detenerlo en el apagado.

```
var Queue = require('simple-redis-safe-work-queue')

var webhookQueueClient = Queue.client('invoke webhook');

var server = Server();

server.listen(8080);

server.post('/some/important/action', function(req, res, next) {
    /// ...

    db.insert(someDoc, function(err) {
        if (err) res.send(err);
        else {
            webhookQueueClient.push({
                url: 'http://example.com',
                method: 'POST',
                json: {
                    a: 1,
                    b: 2
                }
            }, pushedWebhookWork);
        }
    });

    function pushedWebhookWork(err) {
        if (err) res.stats(500).send(err);
        else res.stats(201).send({ok: true});
    }
});

server.once('close', function() {
    webhookQueueClient.quit();
});
```

RABBITMQ para colas distribuidas

- Rabbit-MQ es otro servidor de código abierto que implementa colas persistentes.
- A diferencia de Redis, el único propósito de RabbitMQ es proporcionar una solución de mensajería confiable y escalable con muchas características que no están presentes o son difíciles de implementar en Redis.
- RabbitMQ se puede usar como una plataforma de mensajería.

- RabbitMQ es un servidor que se ejecuta localmente o en algún nodo de la red. Los clientes pueden ser productores de trabajo, consumidores de trabajo o ambos, y hablarán con el servidor usando un protocolo llamado Advanced Messaging Queuing Protocol (AMQP).
- Para Node.js, entre varias bibliotecas que implementan este protocolo.
- Usaremos `ampqlib` por ser el más amigable de usar (<https://www.npmjs.com/package/amqplib>).



Instalando RabbitMQ

- Accede al sitio de RabbitMQ (<http://www.rabbitmq.com/>) e instala el servidor

El Canal

- Para enviar o recibir mensajes, necesitamos establecer una conexión y un canal en el servidor RabbitMQ.
- Vamos a poner el código que es común entre los productores y los consumidores en un archivo común llamado channel.js.
- Este incluye:
 - La creación del canal: `createQueueChannel()`
 - La conexión: `onceConnected()`
 - La creación de la cola una vez creado el canal: `onceChannelCreated()` y `onceQueueCreated()`

channel.js

```
var amqp = require('amqplib/callback_api');

var url = process.env.AMQP_URL ||
'amqp://guest:guest@localhost:5672';

module.exports = createQueueChannel;

function createQueueChannel(queue, cb) {
  amqp.connect(url, onceConnected);

  function onceConnected(err, conn) {
    if (err) {
      cb(err);
    } else {
      console.log('connected');
    }
  }

  function onceConnected(err, conn) {
    if (err) {
      console.error('Error connecting:', err.stack);
    } else {
      console.log('connected');
      conn.createChannel(onceChannelCreated);
    }
  }

  function onceChannelCreated(err, channel) {
    if (err) {
      cb(err);
    } else {
      console.log('channel created');
    }
  }

  function onceQueueCreated(err) {
    if (err) {
      cb(err);
    } else {
      cb(null, channel, conn);
    }
  }
}
```

El Productor - producer.js

```
var Channel = require('./channel');

var queue = 'queue';

Channel(queue, function(err, channel, conn) {
  if (err) {
    console.error(err.stack);
  }
  else {
    console.log('channel and queue created');
    var work = 'make me a sandwich';
    channel.sendToQueue(queue, encode(work), {
      persistent: true
    });
    setImmediate(function() {
      channel.close();
      conn.close();
    });
  }
});

function encode(doc) {
  return new Buffer(JSON.stringify(doc));
}
```

El Consumidor - worker.js

```
var Channel = require('./channel');

var queue = 'queue';

Channel(queue, function(err, channel, conn) {
  if (err) {
    console.error(err.stack);
  }
  else {
    console.log('channel and queue created');
    consume();
  }
}

function consume() {
  channel.get(queue, {}, onConsume);

  function onConsume(err, msg) {
    if (err) {
      console.warn(err.message);
    }
    else if (msg) {
      console.log('consuming %j', msg.content.toString());
      setTimeout(function() {
        channel.ack(msg);
        consume();
      }, 1e3);
    }
    else {
      console.log('no message, waiting...');
      setTimeout(consume, 1e3);
    }
  }
});

});
```

Colas de mensajes Pub/Sub de entrega fiable con Redis

- Extracto del artículo: <http://blog.radiant3.ca/2013/01/03/reliable-delivery-message-queues-with-redis/>
- Cuando el esquema Pub/Sub de redis no es suficiente:
 - Lo que Redis ofrece con Pub/Sub es un modelo de oyente, donde cada suscriptor **recibe los mensajes solo cuando está a la escucha**, no cuando no está conectado.
 - En un entorno de clúster en el que se tienen varios consumidores ejecutándose al mismo tiempo, cada instancia recibiría los mensajes producido en el canal.
 - Queremos asegurarnos de que cualquier mensaje dado se consumió una vez por un consumidor lógico, incluso cuando se ejecutan varias instancias de este componente.
- Se pueden ver un ejemplo de implementación con redis y java en:
 - <https://github.com/davidmarquis/redisq>

5

Node.js for Embedded Systems

Javascript para IoT

(Extracto de <http://embeddednodejs.com/chapters.html>)

- Fácil de aprender
- Ya es un lenguaje común a través de Internet.
- Hay una cantidad enorme de desarrolladores
- Su arquitectura basada en eventos encaja perfectamente con cómo funciona el mundo
- El API de Streams de node.js combina la potencia de los eventos, con la potencia del Unix pipeline

Algunos proyectos

- Espruino:
 - Es un pequeño microcontrolador que ejecuta JavaScript.
 - Necesita muy poco potencia, con una duración de la batería de hasta un año.
 - No requiere un IDE complicado; todo lo que realmente necesita es un terminal.
 - La mejor parte es que, dependiendo de las necesidades de su dispositivo, incluso puede compilar su código JavaScript en C! Desafortunadamente, en la actualidad solo hay un apoyo limitado para eso.
- Tessel 2:
 - Es un tablero del desarrollo que viene con el wi-fi integrado, un puerto de Ethernet, dos puertos del USB, y dos puertos del módulo de Tessel.
 - Cada módulo de Tessel tiene una librería complementaria que se puede descargar con el npm, y hay un montón de tutoriales e instrucciones disponibles en línea para ponerlo en funcionamiento rápidamente.
- Kinoma Create:
 - Denominado el "Juego de construcción de Internet con las herramientas de JavaScript", es una herramienta de prototipos completa con toneladas de capacidades plug-and-play
 - Algunas de las características más atractivas incluyen una pantalla táctil, conexión wi-fi y bluetooth, y una ranura para micro SD.

Node en Raspberry Pi

- El Raspberry Pi es un pequeño ordenador que se puede obtener por unos US \$ 35 y en el que puede ejecutar muchos tipos diferentes de software y construir muchos proyectos.
- Es posible también configurar el Pi como un servidor Node y desplegar aplicaciones basadas en Javascript.

Instalar Node.js

- Es necesario tener configurado el Pi y que este tenga acceso a red
- Es necesario instalar la versión ARM de node

```
> wget http://node-arm.herokuapp.com/node_latest_armhf.deb  
> sudo dpkg -i node_latest_armhf.deb
```

- Una vez hecho esto, comprobar la versión de node

```
> npm -v  
> npm --version
```

- A partir de aquí se desarrollará e instalará la aplicación node como de costumbre

Aplicación en boot

- Definir los elementos que se ejecutan en arranque en /etc/rc.local.
- El shell no tiene la misma ruta de acceso que cuando inicia sesión, por lo que sólo ejecutar "nodo app.js" no será suficiente.
- Lo que mejor funciona, es ejecutar un comando como el usuario predeterminado pi. Dado que ese usuario tiene Node en su ruta.

```
su pi -c 'node /home/pi/server.js < /dev/null &'
```

- Usar una ruta de acceso absoluta a su archivo Node.js sólo para asegurarse.

Wrapping up

- Al ejecutar el siguiente script, intentará instalar Node.js y creará un archivo app.js vacío en su directorio de inicio que se ejecutará cada vez que inicie el Pi.

```
wget http://node-arm.herokuapp.com/node_latest_armhf.deb  
sudo dpkg -i node_latest_armhf.deb  
touch /home/pi/app.js  
su pi -c 'node /home/pi/app.js < /dev/null &'
```

- Más recursos:
 - <http://thisdavej.com/beginners-guide-to-installing-node-js-on-a-raspberry-pi/>

Redis en ARM

- Dado que la versión Redis 4.0 (actualmente release candidate), Redis soporta el procesador ARM en general, y el Raspberry Pi específicamente, como plataforma principal, exactamente como sucede con Linux / x86.
- Esto significa que cada nueva versión de Redis se prueba en el entorno Pi y que llevan una página de documentación actualizada.
- En el siguiente artículo se muestra como configurar redis en un Pi
 - <http://mjavery.blogspot.com.es/2016/05/setting-up-redis-on-raspberry-pi.html>



Analizando una aplicación Node en Pi

- Lee el siguiente artículo que propone crear un Smart tv con Raspberry Pi y Node
 - <https://www.codementor.io/donald/tutorial-build-your-own-smart-tv-using-raspberrypi-nodejs-and-socket-io-8sdcaalbg>
- Descarga la fuente de:
 - <https://github.com/d0nd3r3k/PiR.tv>

6

Profiling de rendimiento

Profiling

- Entender como funciona V8 no es suficiente para escribir aplicaciones NodeJS de alto rendimiento. Es necesario perfilar una aplicación, encontrar el cuello de botella y optimizarlo, sabiendo cómo NodeJS y V8 lo optimizan.
- El objetivo principal de un profiler es medir todos los ticks de CPU en la ejecución de funciones en la aplicación.
- También hay perfiladores de memoria, que se pueden utilizar para encontrar fugas de memoria, pero en este artículo voy a hablar sólo de problemas de rendimiento.
- NodeJS tiene un profiler incorporado, pero con una diferencia: No analiza los archivos de registro como lo hace Google Chrome. En su lugar, recoge toda la información en el archivo de registro.
- Esto significa que se necesita tener alguna herramienta aparte que pueda entender este archivo de registro y proporcionar información legible por humanos.

Proceso de profiling

- Usar "time node <app>" para determinar problemas con el rendimiento
- Usar el profiler de Node: `node --prof <app>`
- Analizar el log con *prof-process*: `node --prof-process <path a logs>`
- Modificar la app para solucionar los problemas de rendimiento



Analizando una aplicación

Script con bottleneck



```
// server.js

"use strict";
const crypto = require('crypto');
function hash(password) {
  const salt = crypto.randomBytes(128).toString('base64');
  const hash = crypto.pbkdf2Sync(password, salt, 10000, 512);
  return hash;
}
// Imagine that loop below is real requests to some route
for (let i = 0; i < 50; i++) {
  hash('random_password');
}
```

Verificar con time

- `time node server.js`



Recopilación de información de ticks en logs

- `node --prof server.js`
 - El log tiene un nombre del tipo: *isolate-*.log*

Analiza los logs

- `node --prof-process isolate-0x101804a00-v8.log > process.txt`

Modifica el script para optimizarlo

Optimizado de código por V8

- En los siguientes artículos se explica cómo V8 optimiza nuestro código a la hora de ejecutar:
 - <https://blog.ghaiklor.com/how-v8-optimises-javascript-code-a0f3bbd46ac9>
 - <https://blog.ghaiklor.com/optimizations-tricks-in-v8-d284b6c8b183>
- Esta información nos puede ayudar a entender mejor cómo funciona internamente V8 a efectos de mejorar el performance de nuestro código.

Claves

- Tratar de minimizar la cantidad de syscalls agrupando / haciendo batching de escritura.
- Considerar la sobrecarga de emitir y borrar los diferentes temporizadores en una aplicación.
- Los perfiladores de CPU dan información útil, pero no cuentan toda la historia.
- Desconfíe de las funcionalidades de ECMAScript de alto nivel, especialmente si no está utilizando el último motor de JavaScript o un transpilador.
- Controlar el árbol de dependencias y compararlas.

Consejos para optimizar las aplicaciones Node.js

➤ Utilizar siempre funciones asíncronas

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

// Performing a blocking I/O
var file = fs.readFileSync('/etc/passwd');
console.log(file);

// Performing a non-blocking I/O
fs.readFile('/etc/passwd', function(err, file) {
  if (err) return err;
  console.log(file);
});
```

Consejos para optimizar las aplicaciones Node.js

- Utilizar módulo `async` para una mejor organización de las funciones (<https://github.com/caolan/async>)

```
var fs = require('fs');
// A common callback hell example
fs.readFile('/etc/passwd', 'utf8', function(passwdErr, passwd) {
  if (passwdErr) return passwdErr;
  fs.readFile('/etc/hosts', 'utf8', function(hostsErr, hosts) {
    if (hostsErr) return hostsErr;
    fs.mkdir(__dirname + '/test', function(dirErr) {
      if (dirErr) return dirErr;
      var data = passwd + hosts;
      fs.writeFile(__dirname + '/test/data', data, 'utf8',
        function(err) {
          console.log('Done!');
        });
    });
  });
});
```

Consejos para optimizar las aplicaciones Node.js

```
// You must install async before: npm install async
var async = require('async');
var fs = require('fs');

async.waterfall([
  function(callback) {
    fs.readFile('/etc/passwd', 'utf8', callback);
  },
  function(passwd, callback) {
    fs.readFile('/etc/hosts', 'utf8', function(err, hosts) {
      if (err) {
        return callback(err);
      }
      var data = passwd + hosts;
      return callback(null, data);
    });
  },
  function(data, callback) {
    fs.mkdir(__dirname + '/test', function(err) {
      if (err) {
        return callback(err);
      }
      return callback(null, data);
    });
  },
  function(data, callback) {
    fs.writeFile(__dirname + '/test/data', data, 'utf8',
    callback);
  }
], function(err) {
  if (err) {
    console.log(err);
    return;
  }
  console.log('Done!');
});
```

Consejos para optimizar las aplicaciones Node.js

- Utilice los generadores ES6 para organizar funciones asíncronas (<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function%2A>)

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

function* fileTask() {
  var passwd =
    yield fs.readFile('/etc/passwd', 'utf8');
  var hosts =
    yield fs.readFile('/etc/hosts', 'utf8');
  var data = passwd + hosts;
  yield fs.mkdir(__dirname + '/test');
  yield fs.writeFile(__dirname + '/test/data', data, 'utf8');
}

var task = fileTask();

task.next(); // Runs the "yield fs.readFile('/etc/passwd', 'utf8')";
task.next(); // Runs the "yield fs.readFile('/etc/hosts', 'utf8')";
task.next(); // Runs the "yield fs.mkdir(__dirname + '/test')";
task.next(); /* Runs the "yield fs.writeFile(__dirname + '/test/data', data, 'utf8');*/
```

Consejos para optimizar las aplicaciones Node.js

- Utilice Node.js únicamente para enviar datos: JSON
- Utilice Nginx o Apache como servidores estáticos
- Evite cookies y sesiones: stateless APIs que proporcionan autenticaciones de token como JWT, OAuth y otros
- Utilizar módulo de clúster para procesamiento en paralelo (<http://nodejs.org/api/cluster.html>)
- Habilitar respuestas de Streaming (<https://nodejs.org/api/stream.html>)
- Utilice siempre la última versión estable

Mejores prácticas de producción: rendimiento y fiabilidad

- Extracto del artículo: <https://expressjs.com/en/advanced/best-practice-performance.html>

En el código (la parte dev)

- Usar compresión gzip

```
var compression = require('compression')
var express = require('express')
var app = express()
app.use(compression())
```

- No utilizar funciones síncronas
- Hacer log correctamente
 - console.log y console.error son síncronos
 - Para debug: módulo debug (<https://www.npmjs.com/package/debug>)
 - Para actividad de la app: módulo winston
(<https://www.npmjs.com/package/winston>)
- Manejar excepciones correctamente
 - No esperar por el evento *uncaughtException*
 - Usar try-catch
 - Usar promesas: módulo bluebird (<http://bluebirdjs.com/docs/api-reference.html>)

Mejores prácticas de producción: rendimiento y fiabilidad

En el entorno / configuración (la parte ops)

- Establecer NODE_ENV a "production"

```
# /etc/init/env.conf
env NODE_ENV=production
```

- Asegurarse de que la aplicación se reinicia automáticamente.

- Gestores de proceso
 - StrongLoop Process Manager (<http://strong-pm.io/>)
 - PM2 (<https://github.com/Unitech/pm2>)
 - npm install pm2 -g
 - pm2 start myApp.js
 - Forever (<https://www.npmjs.com/package/forever>)

- init system
 - systemd (<https://wiki.debian.org/systemd>)
 - Upstart (<http://upstart.ubuntu.com/>)

- Ejecutar la aplicación en un clúster

- Módulo cluster (<https://nodejs.org/dist/latest-v4.x/docs/api/cluster.html>)
- StrongLoop PM (<https://docs.strongloop.com/display/SLC/Clustering>)

- Cachear los resultados de requests

- Vanish (<https://www.varnish-cache.org/>)
- Nginx (<https://www.nginx.com/resources/wiki/start/topics/examples/reverseproxycachingexample>)

- Usar un load balancer: Nginx o HAProxy

<https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts>

- Usar un proxy inverso: Nginx o HAProxy

P

Proyecto Multiproceso con Colas



[...]**netmind**

WeKnowIT

Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es