Modelo asíncrono

Tiempo estimado: 25min

Uno de los aspectos más importantes y que mejor hay que comprender de Node es el modelo asíncrono en el que se basa. Es muy pero que muy importante que preste especial atención a esta lección. Entender el modelo asíncrono es de vital importancia. Sin un conocimiento sólido, el uso de Node se complica muchísimo.

La lección comienza con la presentación del concepto de arquitectura y aquellos tipos en los que se basa Node. A continuación, presentamos la arquitectura de ejecución monohilo. Y finalmente, se hace mucho hincapié en el modelo asíncrono.

Al finalizar la lección, el estudiante sabrá:

- Qué es una arquitectura.
- En qué arquitecturas se apoya Node.
- Cómo funciona la arquitectura monohilo.
- Cómo funciona la arquitectura asíncrona.
- Qué es una función *callback*.

Introducción

Un arquitectura (*architecture*) es un patrón que define cómo estructurar, organizar, diseñar e implementar software. Existe varios tipos de arquitecturas, siendo Node un híbrido de las siguientes:

- Arquitectura de ejecución monohilo.
- Arquitectura asíncrona.
- Arquitectura conducida por eventos.

En Node, las tres se encuentra muy ligadas entre sí. En esta lección, presentamos la arquitectura monohilo y el modelo asíncrono. Dejamos la arquitectura conducida por eventos para una lección posterior.

Arquitectura de ejecución monohilo

Un proceso (*process*) es una instancia en ejecución de un programa o aplicación, cuya ejecución es secuencial. Todo esto significa que como mucho un programa podrá tener una única instrucción en ejecución.

En todo momento, un proceso debe encontrarse en un determinado estado (*state*), el cual indica la situación en que se encuentra, distinguiéndose principalmente los siguientes:

- Nuevo (*new*). El proceso está siendo creado por el sistema operativo.
- En ejecución (*running*). El proceso tiene asignado un procesador y está ejecutando instrucciones actualmente.
- En espera (*waiting*). El proceso está a la espera de que se genere un evento como, por ejemplo, la finalización de una operación de E/S o la lectura del contenido de un archivo.
- Listo (*ready*). El proceso se encuentra en la cola de procesos listos a la espera de que se le asigne un procesador para continuar con su ejecución.
- Terminado (*terminated*). El proceso ha finalizado su ejecución y el sistema operativo está liberando los recursos que le asignó.

Los sistemas operativos de hoy en día son sistemas de multiprogramación, lo que permite que haya varios procesos en ejecución al mismo tiempo. El objetivo de la multiprogramación es tener algún

proceso en ejecución en todo momento, con el objetivo de maximizar el uso de la CPU. Para ello, el sistema operativo distribuye o reparte el uso de CPU en ciclos (*cycles*), períodos de tiempo durante los cuales ejecuta instrucciones de un determinado proceso. Cuando un ciclo finaliza, el sistema coge otro proceso listo y continúa dónde lo dejó antes de ponerlo en una de las colas de espera. Permitiendo así esa sensación de que todos ellos se ejecutan simultáneamente. Obviamente, en todo momento, sólo podrá haber tantos procesos en ejecución como procesadores disponga el sistema.

Cuando un proceso está en ejecución, básicamente puede ocurrir uno de los siguientes acontecimientos:

- Que el proceso ejecute una operación de E/S, por ejemplo, a disco, y entonces tenga que esperar a que la operación termine antes de poder continuar.

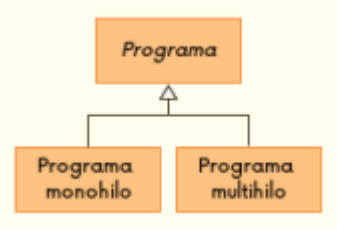
  En este caso, el sistema operativo lo coloca en la cola de espera hasta que la operación de E/S termina. Tras lo cual, moverá el proceso de la cola de espera a la cola de procesos listos para que pueda asignarle de nuevo la CPU y, así, continuar con su ejecución.

- Que el ciclo de procesamiento finalice y el sistema operativo tenga que moverlo de la CPU a la cola de procesos listos para que así otro proceso pueda ejecutarse. Cuando le llegue de nuevo el turno, el sistema operativo lo sacará de la cola de procesos listos y lo volverá a poner en ejecución durante un ciclo de procesamiento.

Independientemente de lo que ocurra, el sistema operativo debe llevar a cabo lo que se conoce formalmente como conmutación de contexto (*context switch*) o conmutación de proceso (*process switch*), operación interna mediante la cual se guarda el estado actual del proceso para que, cuando vuelva a asignarle la CPU, pueda continuar su ejecución donde quedó al perder la CPU. Esta operación es sencillamente un gasto extra adicional necesario para que un sistema pueda soportar la multiprogramación. Cuanto más rápido lo haga el sistema operativo, mejor rendimiento se obtendrá del sistema. Para ello, actualmente el hardware proporciona instrucciones específicas que realizan la operación rápidamente. El componente del sistema operativo que realiza esta operación se conoce formalmente como despachador (*dispatcher*). Y el tiempo que tarda el despachador en realizar la conmutación de contexto, se conoce como latencia del despacho (*dispatch latency*).

Todos los procesos no están en ejecución al mismo tiempo, sólo podrá haber tantos procesos en ejecución como procesadores disponga el sistema. Lo que hace el sistema es dividir el tiempo de asignación de la CPU a los procesos en ciclos y elegir, entre aquellos que están en la cola de procesos listos, a quiénes se les asignará el siguiente ciclo y en qué procesadores. Al componente del sistema operativo, que realiza esta operación, se le conoce formalmente como planificador (*scheduler*). Atendiendo al algoritmo de planificación utilizado por el sistema operativo, este planificador seleccionará unos procesos u otros, entre los listos, para continuar con su ejecución.

Actualmente, los procesos y, por ende, los programas, pueden presentar varios hilos de ejecución. Esto quiere decir que, al igual que un sistema puede estar ejecutando varios procesos al mismo tiempo, dentro de un proceso puede haber varios flujos de ejecución simultáneos. Se utiliza el término hilo (*thread*) o proceso ligero (*lightweight process*) para hacer referencia a un flujo de ejecución dentro de un programa o proceso. Una unidad básica de utilización de CPU. Así pues, se distingue entre programas monohilo y multihilo.



Un programa monohilo (*single-threaded program*) es aquel que presenta un único hilo de ejecución. Mientras que un programa multihilo (*multi-threaded program*) es aquel que presenta varios hilos de ejecución. Los hilos operan de manera muy parecida a los procesos, pero en todo momento son entidades internas a los procesos o programas. El ciclo asignado al proceso se divide en subciclos a repartir entre los hilos.

Un programa monohilo tiene un único hilo de ejecución, por lo tanto, cuando se ejecuta una operación de E/S, el programa queda bloqueado, sin hacer nada, hasta que ésta termine. El sistema operativo lo

llevará a la cola de espera y cuando termine la operación de E/S, lo moverá a la cola de procesos listos para continuar con su ejecución cuando el planificador del sistema operativo vuelva a asignarle la CPU.

En cambio, un programa multihilo tiene una ventaja. Como su ciclo de procesamiento lo reparte entre distintos hilos de ejecución, cada uno de los cuales puede estar ejecutando una cosa distinta, cuando un hilo ejecuta una operación de E/S, sólo *ese* hilo queda a la espera, los demás pueden seguir su ejecución. El proceso no se bloquea, simplemente se bloquea uno de sus flujos de ejecución.

Es importante tener claro que los hilos de ejecución de un proceso se ejecutan de manera muy similar a los procesos en ejecución del sistema. Cuando el sistema operativo asigna un ciclo de procesamiento a un proceso, durante este ciclo se ejecutarán uno o más los hilos del proceso. Por lo que dentro de la ejecución del proceso habrá subciclos para cada uno de los hilos. Está claro, pues, que también existirá el concepto de conmutación, pero esta vez a nivel de hilo. La ventaja de esta conmutación es que es mucho más rápida que la del proceso. También habrá un planificador para seleccionar qué hilo debe ejecutar la CPU durante cada subciclo de procesamiento.

Recopilando, los procesos reciben pequeños períodos de CPU durante los cuales pueden ejecutar sus instrucciones. Si el proceso es multihilo, el sistema repartirá los ciclos de procesamiento del proceso entre sus distintos hilos.

Se ha observado que algunos programas intensivos en E/S como, por ejemplo, las aplicaciones webs o las intensivas en datos, pueden tener un rendimiento mejor en entornos monohilo que multihilo, pero siempre que el programa no se quede bloqueado cuando realiza una operación de E/S. Node es una plataforma monohilo, cuando ejecuta un programa JavaScript lo hace en un único hilo de ejecución. Para evitar los bloqueos de E/S, lo que hace Node es ejecutar funciones de inicio a fin. Si durante la ejecución de la función se invoca una operación de E/S, lo que hace el motor de Node es solicitarla al sistema operativo y registrar esa petición en una cola interna. Cuando el sistema operativo termine la operación de E/S, se lo comunicará a node para que así pueda continuar con lo que resta de la función invocadora. En vez de quedarse parado o bloqueado, lo que hace es recurrir a la cola de operaciones pendientes y las ejecuta y, así, sucesivamente. De esta manera, siempre que tenga algo que ejecutar, lo ejecutará. No dejando que las operaciones de E/S le detengan.

Arquitectura asíncrona

En la arquitectura anterior, se presentó el modelo de ejecución de los procesos. Éste puede ser monohilo o multihilo. Independientemente de cuál se utilice, los procesos deben encontrarse en uno de los posibles estados soportados por el sistema operativo, siendo los ya indicados los más frecuentes.

Recordemos dos estados: listo y en espera. El primero, listo, refleja que el proceso está listo para su ejecución, pero como el sistema dispone de menos procesadores que procesos puede ejecutar, pone en este estado a todos aquellos que podría estar ejecutando, pero no puede por esta falta de recursos. Ahora bien, el segundo, en espera, tiene más miga. Indica que el proceso ejecutó una operación de E/S y hasta que ésta no termine, no puede continuar con su ejecución normal. Por lo que se deposita en una cola aparte para que no pueda ser seleccionado para continuar su ejecución antes de que la operación de E/S termine. Cuando la operación de E/S termine, el sistema cogerá el proceso y lo trasladará de la cola de espera a la cola de procesos listos para que así pueda continuar con su ejecución.

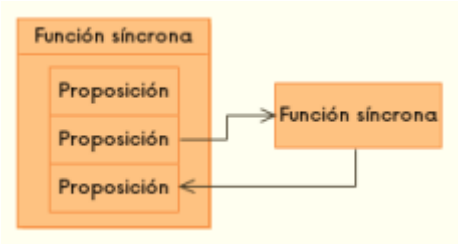
Son muchas las operaciones de E/S que pueden dejar el estado del proceso en espera, por ejemplo, una operación de lectura o escritura de contenido de un archivo o el envío de un mensaje a través de la red. En un sistema multihilo, cuando uno de los hilos se bloquea, debido a una operación de E/S, es sólo ese hilo el que se bloquea; los demás pueden seguir siendo seleccionados para su ejecución porque están listos para continuar. No hay ningún problema al respecto. En cambio, en un sistema monohilo las cosas son muy diferentes. Como el proceso sólo presenta un único hilo de ejecución, ¿el proceso se bloqueará al completo? En efecto. ¿Quiere decir esto que cuando en Node se ejecute una operación de E/S automáticamente se bloqueará? Por suerte, no.

Como Node es monohilo y no desea quedarse bloqueado hasta que la operación de E/S finalice, lo que hace Node es distinguir dos tipos de ejecuciones en el programa: las ejecuciones síncronas y asíncronas.

Cuando el programa invoca una función síncrona, lo que hace el motor de Node es ejecutarla de inicio a fin. Todo el ciclo de procesamiento otorgado por el sistema operativo al intérprete de Node se

utilizará, de inicio a fin, para ejecutar la función síncrona. Si la función no puede finalizar dentro de un ciclo de procesamiento, no pasa nada, cuando vuelva a asignársele un nuevo ciclo, la función continuará. La cuestión es que continúa hasta que finalice. Eso significa que cuando finalice, su funcionalidad se habrá ejecutado completamente. Y si el objeto era devolver un determinado dato, lo devolverá. Está claro que la función puede invocar, a su vez, a otras funciones, tanto síncronas como asíncronas:

- Cuando dentro de una función síncrona se invoca otra síncrona, la segunda se ejecuta de inicio a fin y, cuando finaliza, le devuelve su resultado y el flujo de ejecución a la invocadora para que así pueda continuar con su ejecución.



  Como se puede observar, la ejecución de una función síncrona es de inicio a fin, no hay bloqueos ni nada, extrayéndose todo el juego a la CPU.

- En cambio, cuando en una función síncrona se invoca una asíncrona, las cosas cambian un poco. Generalmente, las funciones asíncronas representan operaciones de E/S como, por ejemplo, ejecutar una consulta contra una base de datos, leer el contenido de un archivo, enviar un mensaje a través de la red, etc. Lo que hace el motor de Node es solicitar la invocación de la operación de E/S al sistema operativo y registrar la petición en una tabla interna. Realizada la petición, continuará con el flujo normal de ejecución hasta que finalice. Sin esperar a que la operación de E/S termine.

Observe la diferencia. Es sutil pero importante. Cuando una función síncrona invoca otra síncrona, la primera se interrumpe, pero no bloquea, hasta que se termina de ejecutar la segunda. En cambio, cuando una función síncrona invoca una asíncrona, lo que sucede es que el motor de Node solicita la operación de E/S al sistema operativo y continúa como si nada, sin esperas, sin bloqueos. ¡node nunca para de trabajar!

Entonces, un aspecto a tener en cuenta es ¿cómo se utiliza los datos devueltos por una invocación asíncrona? Buena pregunta. Si la operación E/S se solicita al sistema operativo y no se espera a que devuelva el resultado, una vez que la función invocadora ha terminado, está claro que esa misma función no podrá utilizarlos. Es parte del juego. Por suerte, JavaScript sale al rescate: al mismo tiempo que se solicita la operación de E/S, se indica una función que debe ejecutarse cuando la operación asíncrona termine para así continuar con la funcionalidad de la función que llevó a cabo el encolamiento. En breve, hacemos más hincapié en esto. Tenga paciencia.

Operaciones de E/S

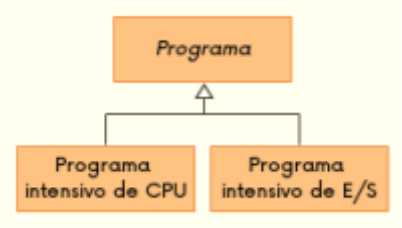
Un ordenador tiene básicamente dos tipos de tareas, la entrada/salida (E/S) y el procesamiento. El sistema de E/S (*I/O system*) es el componente del sistema operativo que se encarga de administrar e interactuar con los dispositivos o periféricos como, por ejemplo, el disco, la tarjeta de red, el teclado, el ratón o la pantalla. Teniendo en cuenta esto, podemos decir que un ordenador tiene básicamente dos tareas principales, la E/S y el procesamiento. Es más, se ha observado que en muchos casos la tarea principal es la E/S y, en otros, el procesamiento.

Una operación de E/S (*I/O operation*) es la que realiza una tarea sobre un dispositivo de E/S como, por ejemplo, la lectura de un bloque de disco o el envío de datos a través de la tarjeta de red. Las operaciones de E/S son mucho más lentas que las de procesamiento. Groso modo, estas operaciones se dividen en tres operaciones: la de inicio, la de realización y la de fin. Mediante la operación de inicio, solicitamos algo al dispositivo de E/S. Una vez realizada, el programa queda a la espera de que el dispositivo lleve a cabo lo solicitado. Una vez el periférico finaliza el trabajo solicitado, comunica el resultado mediante la operación de finalización. Tras esta operación, el programa puede continuar con su trabajo. Se desbloquea y continúa donde se quedó, usando los datos devueltos por el sistema de E/S.

Debido a la latencia inherente de las operaciones de E/S, la E/S es un factor muy importante que

reduce el rendimiento de un sistema. Y debido a que en muchos casos los programas deben esperar a que las operaciones de E/S finalicen para poder continuar, lo que sucede es que la operación se bloquea a la espera de que la E/S finalice.

Desde mucho tiempo atrás, se ha realizado estudios sobre el comportamiento de los procesos y, por ende, de los programas o aplicaciones. Se ha observado que los programas tienden a alterar constantemente entre ráfagas de CPU y ráfagas de E/S. Atendiendo a la duración de los dos tipos de ráfagas, se ha clasificado los programas entre intensivos de CPU e intensivos de E/S.



Un programa intensivo de CPU (*CPU-intensive program*) es aquel en el que predominan las ráfagas de CPU con respecto a las de E/S. Suelen ser programas que realizan principalmente cálculos y pocos accesos a los recursos de E/S. En cambio, un programa intensivo de E/S (*I/O-intensive program*) es aquel en el que predominan las ráfagas de E/S con respecto a las de CPU, son generalmente intensivos en datos como, por ejemplo, las aplicaciones webs o las aplicaciones de *streaming*. Node presenta muy buen rendimiento en los programas intensivos de E/S.

Estilo de paso de continuación

Para comprender el mecanismo asíncrono de Node, básicamente hay que comprender dos estilos de programación, el directo y el de paso de continuación.

En el estilo directo (*direct style*), el programa se ejecuta secuencialmente, es decir, se ejecutan las líneas de código una detrás de otra, y no se pasa a la siguiente hasta que no se ha terminado con la anterior. El código del programa se puede dividir en bloques, fragmentos u operaciones. Pero en todo momento, se ejecutará uno detrás de otro. Si la función invoca una operación asíncrona, el sistema espera a que la E/S termine para continuar. En definitiva, es bloqueador. Este estilo es muy poco recomendable en los programas intensivos de E/S porque se desperdicia recursos y aumenta la latencia. Pero su ventaja es que es muy fácil de entender y aprender.

El estilo de paso de continuación (*continuation-passing style*, CPS) es más difícil de entender y usar, pero no mucho más. Sólo hay que comprenderlo. Bajo este estilo, el programa se tiene que dividir en bloques. Cada uno de ellos tendrá una funcionalidad clara y bien definida. Los bloques se ejecutarán uno detrás de otro. La diferencia con respecto al estilo directo está relacionada con las operaciones de E/S. Cuando se ejecuta una operación de E/S, en el estilo directo, la aplicación espera a que termine, antes de continuar. Podemos decir, sin miedo a equivocarnos, que la aplicación se queda bloqueada. En cambio, en el modelo CPS, las cosas son distintas. Cuando se ejecuta una operación de E/S, lo que se hace es indicar, junto a la invocación asíncrona, la operación que debe ejecutarse cuando termine. Es no bloqueador.

Así pues, un programa escrito bajo CPS, se divide, por un lado, en operaciones síncronas, cuyas líneas se ejecutan secuencialmente, una detrás de otra. Y por otro lado, en operaciones asíncronas, aquellas que se asocian a operaciones de E/S y, por lo tanto, reciben un argumento extra, el bloque que debe invocarse cuando finalice la E/S para continuar así con el trabajo secuencialmente.

Las operaciones que contiene bloques del programa, que deben ejecutarse cuando termina la operación de E/S, se conocen formalmente como funciones de continuación (*continuation functions*) o callbacks. Contienen el código del siguiente bloque operativo de trabajo, el cual debe ejecutarse tras la finalización de la operación de E/S que estamos pendientes de que termine. Generalmente, porque necesitamos los datos que devolverá la E/S.

Una de las principales ventajas del estilo CPS, es que se puede realizar otras tareas mientras se está a la espera de que la operación de E/S finalice. Por esta razón, se dice que es no bloqueador (*non-blocking*). Observe que el concepto de si algo es bloqueador, o no lo es, está relacionado con cómo actúa el sistema cuando se realiza una operación de E/S. Si el programa se queda bloqueado hasta que termine la operación de E/S, se dice que es bloqueador. Si no lo hace, es no bloqueador. Node es no bloqueador, porque no espera a que termine, usando las *callbacks* para indicar dónde continuar cuando la operación de E/S termine.

En este punto, vamos a hacer un pequeño resumen para ubicarnos mejor en el mundo Node.

Por un lado, tenemos que Node es monohilo. En todo momento, sólo hay un hilo de ejecución en el programa. Esto es fácil de comprender.

Por otro lado, y debido a la naturaleza monohilo de Node, hay que sacarle provecho al único hilo de ejecución. Para ello, se utiliza un sistema asíncrono de ejecución. Existe dos tipos de operaciones, las síncronas y las asíncronas. Las síncronas se ejecutan siempre de inicio a fin. Cada vez que el motor de Node ejecuta una función síncrona (*synchornous function*), aquella que no lleva a cabo ninguna operación de E/S, lo hace línea a línea, de inicio a fin, sin bloqueos. Si termina el ciclo de procesamiento asignado por el sistema operativo y no ha finalizado la operación, no pasa nada. El sistema operativo lo llevará a su cola de procesos listos y cuando le llegue de nuevo el momento le asignará la CPU. Y entonces, el motor de Node continuará en el punto que se quedó de la operación síncrona. Así, hasta terminarla.

En cambio, cuando el motor se encuentra con una función asíncrona (*asynchronous function*), aquella que contiene en su cuerpo una operación de E/S, el funcionamiento es muy similar. Ejecutará el cuerpo de la función de inicio a fin. Pero cuidado, cuando se encuentre con la operación de E/S, le solicitará al sistema de E/S del sistema operativo su ejecución, pero no esperará a que termine. Continuará con lo que le resta de función.

Cuando el sistema operativo termina la operación de E/S, se lo indica a node. Entonces node va a la tabla donde tiene registradas las operaciones de E/S solicitadas y en espera. Coge el registro de la operación que acaba de terminar y añade su *callback* a su cola de espera para que así se pueda continuar con su ejecución. Cuando node alcanza el final de la función que comenzó, va a la cola y extrae su siguiente tarea a trabajar. Generalmente, la función *callback* de una operación de E/S.

Las funciones asíncronas dividen su cuerpo, pues, en bloques. Está claro que cuando nos encontramos con un bloque asíncrono, el siguiente generalmente no puede continuar hasta que la E/S le proporcione los datos que necesita. Por ejemplo, supongamos que la operación de E/S tiene como objeto obtener una fila de una base de datos. Es inevitable que lo que resta de la función no se pueda llevar a cabo hasta que se disponga de ese dato. Entonces lo que se hace es coger el bloque que depende del dato y ubicarlo en una función JavaScript, la cual se pasará a la operación de E/S. A esta función que contiene un bloque de funcionalidad que debe ejecutarse al finalizar la operación de E/S se le conoce formalmente como callback. Cuando la E/S termine, cogerá el dato, invocará la función *callback* pasándole el dato como argumento, continuando así con el trabajo asignado a la función asíncrona.

Así pues, podemos decir que cuando una función síncrona finaliza su ejecución, su funcionalidad se ha ejecutado completamente. No podemos decir lo mismo de las funciones asíncronas. Han finalizado su cuerpo, pero no todos sus bloques. Los que dependen de operaciones de E/S no se han ejecutado todavía. Sólo se han encolado para su ejecución. Están pendientes. No olvide esto: síncrono, todo finalizado y completado; asíncrono, finalizado pero no completado.

Veamos esto último mediante un ejemplo. Supongamos que deseamos mostrar al usuario el contenido de un archivo. La lectura de un archivo se puede hacer mediante la función asíncrona readFile() del módulo fs que viene de fábrica con Node. Imaginemos que deseamos mostrar un mensaje al usuario, antes de los datos como, por ejemplo, INICIO. Y otro después de mostrar el contenido: FIN. En este caso, nuestra tarea se divide en tres bloques:

1. Mostrar el mensaje INICIO.

2. Mostrar el contenido del archivo.
3. Mostrar el mensaje FIN.

Sabemos que la lectura del contenido es asíncrona. Por lo que habrá que pasarle una función *callback* a la función readFile() para que, por una parte, nos pase el contenido del archivo y, por otra parte, la función *callback* pueda mostrar el contenido y, finalmente, el mensaje FIN. ¿Cómo implementaríamos esto? Fácil:

```
console.log("INICIO");

fs.readFile("/ruta/al/archivo.txt", function callback(err, data) {
  if (err) return console.error(err);

  console.log(data.toString());
  console.log("FIN");
});
```

Así pues, si tenemos que el contenido del archivo es: ¡Hola, Mundo! La salida será como sigue:

```
INICIO
¡Hola, Mundo!
FIN
```

¿Qué ocurre si sacamos el mensaje FIN y lo ponemos justo después de invocar la función asíncrona readFile()? O sea, si tenemos lo siguiente:

```
console.log("INICIO");

fs.readFile("/ruta/al/archivo.txt", function callback(err, data) {
  if (err) return console.error(err);
  console.log(data.toString());
});

console.log("FIN");
```

La salida sería:

```
INICIO
FIN
¡Hola, Mundo!
```

¿Por qué? Recordémoslo por última vez. Porque las invocaciones de operaciones asíncronas se encolan sin esperar a que las operaciones de E/S adjuntas finalizan. Por lo que nunca se ejecutan hasta que la función actual ha finalizado y también lo ha hecho la operación de E/S. Una vez registrada en la cola, el flujo de ejecución sigue con la proposición que le sigue. Así pues, el flujo anterior se resumiría en:

1. Mostrar el mensaje INICIO.
2. Solicitar la operación de E/S al sistema operativo y registrar la función callback para cuando se reciba la terminación de la operación de E/S.
3. Mostrar el mensaje FIN.

Como nosotros queremos que el mensaje FIN se muestre después del contenido del archivo, debemos hacerlo en la lógica de la función *callback*. No hay que añadirlo después de la invocación a la función asíncrona, sino que hay que hacerlo en la función *callback*.

Ahora, veamos cómo sería el mismo código mediante una operación de E/S síncrona bloqueadora:

```
console.log("INICIO");
console.log(fs.readFileSync("/ruta/al/archivo.txt").toString());
console.log("FIN");
```

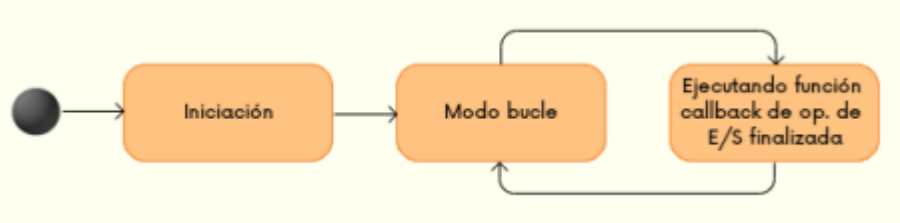
El problema de esta segunda versión, es que la operación fs.readFileSync() es una implementación síncrona de la función asíncrona fs.readFile(). Este tipo de funciones lo que hace es invocar la función asíncrona y esperar a que ésta termine. En resumen, bloquean el flujo de ejecución de Node hasta que la E/S finalice. Esto, obviamente, reducirá el rendimiento y aumentará la latencia. En muchas ocasiones, este comportamiento es aceptable. Pero siempre que sea posible, hay que invocar las funciones asíncronas como tales.

Cuando se invoca una función, hay que saber si es síncrona o asíncrona. Básicamente, si la función devuelve una promesa o contiene una función *callback*, la función será asíncrona; en otro caso, generalmente, será síncrona. Es importante saberlo porque cuando es asíncrona, el código del siguiente bloque de operación debe ejecutarse una vez ha finalizado la operación de E/S.

Flujo de ejecución del programa

Una vez tenemos claro cómo funciona el modelo asíncrono utilizado en Node, vamos a presentar el flujo de ejecución de un programa o *script* escrito en JavaScript para su ejecución en Node.

A node hay que pasarle el archivo principal o punto de entrada de la aplicación o *script*. Su código se ejecutará síncronamente, es decir, de inicio a fin. Cuando se encuentre con una proposición que invoca una función asíncrona, el motor la solicitará y continuará sin esperar el resultado. Al finalizar la ejecución del archivo principal, entonces entrará en modo bucle y comenzará a ejecutar las operaciones registradas en la cola, una a una, de inicio a fin.



La cola se irá llenando de funciones *callbacks* asociadas a operaciones de E/S finalizadas. Y lo hará a medida que el sistema de E/S indique que ha terminado.

