Módulo fs

Tiempo estimado: 20min

En este punto del programa, vamos a hacer un pequeño parón con objeto de presentar un módulo integrado a través del cual mostrar cómo se puede trabajar con Node síncrona y asíncronamente. Para ello, hemos elegido el módulo fs que proporciona una API con la que acceder al sistema de ficheros. La idea es consolidar mejor el modelo asíncrono inherente de Node mediante un módulo muy utilizado e integrado en el motor de Node.

La lección comienza recordando los conceptos de módulo y sistema de ficheros. A continuación, se muestra cómo acceder a los metadatos de una entrada del sistema de ficheros. Seguimos con secciones específicas para los directorios y los archivos. Y finalmente, presentamos cómo renombrar entradas.

Al finalizar la lección, el estudiante sabrá:

- Cómo acceder a los metadatos de las entradas del sistema de ficheros.
- Cómo leer, escribir y suprimir archivos.
- Cómo leer, crear y suprimir directorios.
- Cómo renombrar archivos, directorios y enlaces.

Introducción

Recordemos que un módulo (*module*) es un archivo de código JavaScript que implementa una determinada funcionalidad reutilizable, clara y bien definida. Atendiendo a si el módulo viene de fábrica con Node, los módulos se clasifican en integrados o de sistema y personalizados o de usuario. Realmente, todos los módulos *no* tienen que estar escritos en JavaScript, también pueden ser nativos, estar escritos en C/C++, proporcionando funcionalidad que de otra manera no estaría disponible. El módulo fs es un ejemplo de este tipo de módulos.

El sistema de ficheros (*file system*) es el componente del sistema operativo que se encarga de administrar los ficheros. Se encarga de las operaciones de E/S con los dispositivos que almacenan los ficheros, ya sea para leer o escribir su contenido, así como para leer o escribir sus metadatos.

El módulo fs se encarga de interactuar con el sistema de ficheros, de manera asíncrona. Proporciona funciones asíncronas para leer y escribir archivos, así como sus metadatos. Aunque dispone de versiones síncronas de las anteriores, las cuales por lo tanto serán bloqueadoras. Siempre que sea posible, se recomienda utilizar las asíncronas porque mejoran el rendimiento del sistema en su conjunto.

Vamos a presentar las funciones más utilizadas. La lista completa se puede encontrar en nodejs.org/dist/latest/docs/api/fs.html. Échele un vistazo cuando termine la lección.

Acceso a metadatos

Como ya sabemos, un metadato (*metadata*) es un dato que describe otro dato. Por ejemplo, el archivo package.json es un archivo de metadatos. Los archivos, directorios y enlaces almacenados en un sistema de ficheros también presentan metadatos, además de contenido. Entre otros encontramos: el i-nodo, el identificador de su propietario, la fecha de creación, la fecha de modificación, etc.

Esta información se puede obtener mediante las funciones stat(), lstat() y fstat():

```
function stat(path, callback)
function statSync(path) : Stats
function lstat(path, callback)
function lstatSync(path) : Stats
function fstat(fd, callback)
function fstatSync(fd) : Stats
```


Parámetro	Tipo de datos	Descripción
path	string	Ruta de la entrada del sistema de ficheros a consultar.
fd	number	Descriptor de la entrada del sistema de ficheros a consultar.
callback	function	Función a invocar cuando la operación de E/S haya terminado: fn(error, stats).

Observe que todas las funciones disponen de una versión asíncrona y otra síncrona. La primera, obviamente, *no* es bloqueadora; mientras que la segunda lo es. Recordemos que el concepto bloqueador lo que indica es si el flujo normal de ejecución se detiene cuando se realiza la operación de E/S. Cuando hay bloqueo, se reduce el rendimiento. Pero, es muy útil en determinadas ocasiones. Por ejemplo, si estamos trabajando con el intérprete interactivo, usaremos siempre que sea posible versiones síncronas porque hacen la sesión más fácil.

¿Cuál es la diferencia entre las tres funciones? stat() devuelve un objeto Stats con los metadatos de la entrada consultada del sistema de ficheros. lstat() hace lo mismo pero se usa para consultar enlaces. Finalmente, fstat() se utiliza cuando tenemos un descriptor de archivo, en vez de su ruta. Recordemos que un descriptor de archivo (*file descriptor*) no es más un valor numérico que identifica una apertura de archivo a través de la cual poder realizar operaciones de L/E con el archivo.

Las funciones devuelven una instancia de la clase fs.Stats que contiene los metadatos. Por un lado, este objeto tiene métodos con los que comprobar el tipo de entrada:

```
isFile() : boolean
isDirectory() : boolean
isBlockDevice() : boolean
isCharacterDevice() : boolean
isFIFO() : boolean
isSocket() : boolean
isSymbolicLink() : boolean
```

Los métodos son autoexplicativos. Por lo que no haremos más hincapié en ellos. Sólo mencionar que el método isSymbolicLink() sólo se puede usar cuando el objeto lo devuelve lstat().

Por otra parte, la instancia Stats dispone de varias propiedades con las que obtener los metadatos, entre las cuales encontramos:

- ino (number). I-nodo de la entrada.
- mode (number). Permisos de acceso al archivo.
- uid (number). Identificador del usuario propietario.
- gid (number). Identificador del grupo propietario.
- size (number). Tamaño en bytes.
- blksize (number). Tamaño de bloque en bytes.
- blocks (number). Bloques asignados a la entrada.
- atime (Date). Fecha y hora del último acceso.
- mtime (Date). Fecha y hora de la última modificación.
- ctime (Date). Fecha y hora del último cambio del i-nodo.
- birthtime (Date). Fecha y hora de creación.

Presentados los jugadores, comencemos el juego. Primero, veamos cómo usar, por ejemplo, el intérprete interactivo para obtener los metadatos de un archivo, mediante la versión síncrona y bloqueadora:

```
> fs.statSync("archivo.txt")
{ dev: 64514,
  mode: 33204,
  nlink: 1,
  uid: 1000,
  gid: 1000,
  rdev: 0,
  blksize: 4096,
  ino: 1326404,
  size: 21,
```



```
  blocks: 8,
  atime: 2016-10-22T11:16:15.030Z,
  mtime: 2016-10-22T11:16:12.518Z,
  ctime: 2016-10-22T11:16:12.518Z,
  birthtime: 2016-10-22T11:16:12.518Z }
>
```

Y ahora, veámoslo mediante la versión asíncrona y, por qué no, también en el intérprete interactivo:

```
> fs.stat("archivo.txt", function(error, stats) {
... if (error) return console.error(error);
... console.dir(stats);
... })
undefined
> { dev: 64514,
    mode: 33204,
    nlink: 1,
    uid: 1000,
    gid: 1000,
    rdev: 0,
    blksize: 4096,
    ino: 1326404,
    size: 21,
    blocks: 8,
    atime: 2016-10-22T11:16:15.030Z,
    mtime: 2016-10-22T11:16:12.518Z,
    ctime: 2016-10-22T11:16:12.518Z,
    birthtime: 2016-10-22T11:16:12.518Z }
```

Y como no podía ser de otra manera, recordemos, por *enésima* vez cómo funciona la asincronía. La función asíncrona stat() solicita al sistema de ficheros que le proporcione los metadatos del archivo especificado, mediante una operación de E/S. Node en vez de quedarse esperando, como si no tuviera nada más importante que hacer, regrista la función *callback* que se adjunta en la función y continúa con su flujo de ejecución. En este caso, no tiene nada más que hacer y el intérprete muestra el *prompt*.

Mientras el intérprete espera que introduzcamos una nueva proposición a ejecutar, el sistema de E/S termina la operación de E/S, informa a node de esto y le pasa los datos solicitados. Entonces Node, coge la función *callback* adjuntada y la envía a la cola de espera. Como Node está en modo bucle y no está haciendo nada, va a la cola de espera, extrae la función *callback* y la invoca. La cual no hace más que mostrar el contenido del objeto Stats devuelto por el sistema de ficheros. De ahí que aparezca justo después del *prompt* del intérprete.

Directorios

Un directorio (*directory*) o carpeta (*folder*) no es más que un tipo de entrada especial del sistema de ficheros que actúa como contenedor de otras entradas. Se utilizan, como ya todos sabemos, para organizar el contenido del sistema de ficheros.

Lectura de entradas

Para leer las entradas de un directorio, hay que utilizar la función readdir() del módulo fs:

```
function readdir(path, callback)
function readdir(path, opts, callback)
function readdirSync(path) : string[]
function readdirSync(path, opts) : string[]
```

Parámetro	Tipo de datos	Descripción
path	string	Ruta al directorio a consultar.
opts	object	Opciones de consulta: • encoding (string). Codificación de las cadenas devueltas por la función. Valor predeterminado: utf8.
callback	function	Función a invocar cuando la operación de E/S haya terminado: fn(error, entries).

La función también dispone de versión síncrona bloqueadora y asíncrona no bloqueadora. Observe que la función *callback* difiere su signatura de la presentada en la función stat(). stat() pasa como segundo argumento una instancia de la clase fs.Stats mientras que readdir() un *array* de cadenas que contiene los nombres de las entradas del directorio.

He aquí un ejemplo de invocación de la versión síncrona que sirve para mostrar el resultado devuelto por el sistema de ficheros a la operación de E/S solicitada por la función síncrona readdirSync():

```
> fs.readdirSync(".")
[ '.editorconfig',
  '.eslintignore',
  '.eslintrc',
  '.git',
  '.gitignore',
  '.travis.yml',
  'Justo.js',
  'README.md',
  'build',
  'dist',
  'index.js',
  'lib',
  'node_modules',
  'package.json',
  'template',
  'test' ]
>
```

Creación de directorios

Para crear un directorio, se puede usar la función mkdir() del módulo fs:

```
function mkdir(path, callback)
function mkdir(path, mode, callback)
function mkdirSync(path)
function mkdirSync(path, mode)
```

Parámetro	Tipo de datos	Descripción
path	string	Ruta del directorio a crear.
mode	number	Permisos del directorio a crear. Valor predeterminado. 0o777.
callback	function	Función a invocar cuando la operación de E/S haya terminado: fn(error).

Esta función muestra un aspecto importante de las funciones *callback*. Observe que las funciones que hemos visto hasta ahora tienen como primer parámetro error. Con él, lo que hace el sistema de E/S es indicar si se ha producido un error en la operación de E/S. Si así es, contienen el error en cuestión; en otro caso, null o undefined.

Por otra parte, observe que las anteriores tenían un segundo parámetro, aquel a través del cual reciben los datos solicitados. Esta función no tiene. ¿Por qué? Porque mkdir() no está solicitando ningún dato, sino que el sistema de ficheros cree un directorio. O lo hace o no lo hace. Punto. Esto nos permite ver que las operaciones de E/S no siempre tienen como objeto esperar un dato. Pero aún así, la función asíncrona debe esperar a que el sistema de E/S termine la operación, le indique que ha terminado y si lo ha hecho en error o con éxito.

La cuestión es que el sistema de E/S tarda tiempo en terminar la operación solicitada. Y Node no quiere desechar recursos quedándose bloqueado sin hacer nada. Decide seguir haciendo cosas. Cuando el sistema le indique que ha terminado, encolará la función de *callback* y cuando pueda la ejecutará. Así siempre está trabajando y sacando provecho de los recursos del sistema.

Supresión de directorios

Para suprimir un directorio, el cual debe encontrarse vacío, se puede usar la función rmdir():

```
rmdir(path, callback)
rmdirSync(path)
```


Parámetro	Tipo de datos	Descripción
path	string	Ruta al directorio a suprimir.
callback	function	Función a invocar cuando la operación de E/S haya terminado: fn(error).

Archivos

Un archivo (*file*) o fichero (*file*) es un recurso del sistema de ficheros que contiene datos como, por ejemplo, un documento HTML, un archivo JavaScript, un archivo PDF, una imagen, etc.

Lectura del contenido de un archivo

Para leer el contenido de un archivo, se puede utilizar la función readFile():

```
readFile(path, callback)
readFile(path, opts, callback)
readFileSync(path) : Buffer
readFileSync(path, opts) : Buffer | string
```

Parámetro	Tipo de datos	Descripción
path	string	Ruta al archivo a leer.
opts	object	Opciones de lectura: • encoding (string). Codificación del contenido. Valor predeterminado. null.
callback	function	Función a invocar cuando la operación de E/S haya terminado: fn(error, data).

De manera predeterminada, la función devuelve los datos en forma binaria en un búfer. Si sabemos que el contenido es textual, podemos solicitar que devuelva una cadena mediante la opción encoding a utf8.

Ejemplo:

```
> fs.readFileSync(".editorconfig")
<Buffer 5b 2a 5d 0a 69 6e 64 65 6e 74 5f 73 74 79 6c 65 20 3d 20 73 70 61 63 65 0a 69 6e 64
65 6e 74 5f 73 69 7a 65 20 3d 20 32 0a 65 6e 64 5f 6f 66 5f 6c 69 ... >
> fs.readFileSync(".editorconfig", {encoding: "utf8"})
'[*]\nindent_style = space\nindent_size = 2\nend_of_line = lf\n'
> fs.readFileSync(".editorconfig").toString()
'[*]\nindent_style = space\nindent_size = 2\nend_of_line = lf\n'
>
```

Modificación del contenido de un archivo

Para reemplazar el contenido de un archivo por otro, hay que usar la función writeFile():

```
function writeFile(path, data, callback)
function writeFile(path, data, opts, callback)
function writeFileSync(path, data)
function writeFileSync(path, data, opts)
```

Parámetro	Tipo de datos	Descripción
path	string	Ruta al archivo a sobrescribir.
data	string o Buffer	Contenido a escribir.
opts	object	Opciones de escritura: • encoding (string). Codificación en la que escribir el contenido. Valor predeterminado: utf8. • mode (number). Permisos. Valor predeterminado: 0o666.
callback	function	Función a invocar cuando la operación de E/S haya terminado: fn(error).

Añadidura de contenido a un archivo

Otra operación de escritura es la añadidura de contenido al final del archivo. Para este fin, disponemos de la función appendFile():

```
function appendFile(path, data, callback)
function appendFile(path, data, opts, callback)
function appendFileSync(path, data)
function appendFileSync(path, data, opts)
```

Parámetro	Tipo de datos	Descripción
path	string	Ruta al archivo a actualizar.
data	string o Buffer	Contenido a añadir.
opts	object	Opciones de escritura. Ídem a writeFile().
callback	function	Función a invocar cuando la operación de E/S haya terminado: fn(error).

Supresión de archivo

Para suprimir un archivo, hay que utilizar la función unlink():

```
function unlink(path, callback)
function unlinkSync(path)
```

Parámetro	Tipo de datos	Descripción
path	string	Ruta al archivo a suprimir.
callback	function	Función a invocar cuando la operación de E/S haya terminado: fn(error).

Renombramiento de entrada

Para renombrar una entrada del sistema de ficheros como un directorio, archivo o enlace, podemos utilizar la función rename():

```
function rename(oldPath, newPath, callback)
function renameSync(oldPath, newPath)
```

Parámetro	Tipo de datos	Descripción
oldPath	string	Ruta a la entrada a renombrar.
newPath	string	Nueva ruta de la entrada.
callback	function	Función a invocar cuando la operación de E/S haya terminado: fn(error).

