Ouvrir ou fermer un fichier

Temps de lecture : 7 minutes

Utilisation du module `fs`

Le module `fs` est un module `core`, il peut donc s'importer de cette manière :

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

Méthodes synchrones et asynchrones

Toutes les opérations sur les systèmes de fichier ont une méthode sur l'objet `fs`.

Chaque opération peut être réalisée de manière **synchrone ou asynchrone** et chaque opération a donc deux méthodes pour chaque forme d'exécution.

Les méthodes synchrones **sont exécutées sur le `thread` principal, elle bloque donc l'`event loop`**.

Elles ne sont à utiliser que lorsqu'il est absolument nécessaire que l'opération sur le système de fichier se passe à un moment précis et qu'il faut que toutes les autres opérations attendent.

Nous verrons plusieurs exemples. Mais nous pouvons déjà citer : chargement d'un fichier de configuration, chargement de certificats SSL pour lancer un serveur HTTPS.

Pour **toutes les autres opérations il faut donc utiliser la forme asynchrone qui ne bloquera pas le `thread` d'exécution du processus `Node.js` de votre script ou de votre serveur Web**.

Avancé : méthodes asynchrones et utilisation du `thread pool`

Nous allons revoir encore cette notion car cela peut avoir des implications très importantes sur les performances de votre application.

Comme nous l'avions vu, lorsque `Node.js`, dispose d'un système asynchrone, à travers `lib uv` ou autre, il va l'utiliser.

Si il n'existe pas de système asynchrone au niveau du système d'exploitation, il va **utiliser le `thread pool` de `libuv` pour simuler l'asynchrone**.

Pourquoi ? Car qu'est ce que veut éviter `Node.js` à tout pris ?

Si vous avez bien suivi, il veut éviter de bloquer l'`event loop` qui est sur un **unique thread**.

Pour les systèmes qui ne sont pas asynchrones, `Node.js` va donc demander à `libuv` un `thread` libre dans le `thread pool`.

Rappelez vous par défaut il y a 4 `thread` par processus `Node.js`.

Ce `thread` va donc s'occuper de travailler sur l'opération de manière synchrone mais de son côté, et il préviendra `libuv` quand il aura terminé par un `event`, la fonction de `callback` liée sera alors exécutée.

La liste exacte des modules qui utilisent la `thread pool` car il n'existe pas de système bas niveau asynchrone est : `fs` sauf les méthodes synchrones, certaines fonctions du module `crypto`, la fonction de résolution `dns.lookup()` et tout le module de compression `zlib`.

Que se passe t-il si vous avez plus de quatre `thread` occupés par ces tâches ? Les tâches seront mises en file d'attente par `libuv` et il faudra attendre qu'un `thread` se libère. L'utilisateur devra attendre.

Gestion des erreurs

Pour les versions **synchrones** des méthodes, les exceptions sont `throw` dès qu'elles surviennent.

Il est recommandé d'utiliser `try / catch` pour les gérer.

Pour les versions `asynchrones` des méthodes, si une exception a lieu, **elle est passée à la fonction de `callback` en premier argument**.

Si il n'y a pas d'erreur, le premier argument vaudra `null` ou `undefined`.

Voici un exemple :
https://repl.it/@dymafr/node-c4-l2-1

Ordre des opérations asynchrones sur les fichiers

Les opérations asynchrones sur le système de fichier sont confiés à la `threadpool` et sont effectuées en parallèle sur quatre `thread`. Il n'y a donc absolument aucune garantie quant à l'ordre d'exécution.

Prenons le code suivant :

```
fs.stat('./inconnu', err => console.log(err));
fs.stat('./inconnu2', err => console.log(err));
```

Les deux opérations s'exécuteront dans un ordre non garanti.

Cela peut conduire à des `race conditions`. Par exemple, si vous voulez ouvrir puis lire un fichier, si vous déclarez les méthodes comme cela, l'opération de lecture pourrait survenir avant l'opération d'ouverture et vous obtiendrez une erreur.

Il faut donc utiliser les `callbacks` pour garantir l'ordre d'exécution :

```
fs.stat('./inconnu', err => {
  console.log(err);
  fs.stat('./inconnu2', err => console.log(err));
});
```

Ouvrir et fermer un fichier

Pour ouvrir un fichier de manière asynchrone puis le fermer il suffit de faire :

```
fs.open('file.txt', 'r', (err, fd) => {
  if (err) throw err;
  fs.close(fd, (err) => {
    if (err) throw err;
  });
});
```

En premier paramètre, il faut passer le `path` en relatif ou en absolu, vous pouvez vous aider de la variable globale `__dirname`.

Les descripteurs de fichiers

Un élément très important pour le module `fs` et le système de fichiers, est le `file descriptor` **ou** `fd`.

Il s'agit d'un **identifiant unique numérique** permettant de suivre les fichiers traités par le module. Cet identifiant est ensuite utilisé par le système de fichier de l'environnement système.

En fait, il s'agit d'une abstraction de `Node.js` pour ne pas avoir à interagir vous même avec les différents systèmes de suivi des fichiers suivant les systèmes d'exploitation.

La méthode `fs.open()`

Comme nous l'avons vu plus haut, la méthode `fs.open()` retourne en deuxième argument un `file descriptor`.

En fait, cette méthode permet de **créer un nouvel identifiant** `fs` pour le fichier qui va être traité.

Une fois cet identifiant attribué vous pourrez effectuer toutes les opérations que vous voulez sur le fichier : lecture, écriture etc.

La méthode `fs.open()` peut prendre entre deux et quatre arguments.

```
open(path, flags, mode, callback)
```

Le `path` peut être relatif ou absolu comme nous l'avons vu. C'est un argument obligatoire.

Le deuxième argument `flags` permet de spécifier le comportement du fichier une fois l'identifiant attribué. Nous allons y revenir en détails. Mais par défaut le `flag` sera défini à `r` ce qui signifie que le fichier ne pourra qu'être lu et qu'une erreur sera `throw` si il n'existe pas.

Le troisième argument permet de définir les permissions du fichier mais **uniquement si le fichier est créé**. Par défaut les permissions sont définis à `0666`, c'est à dire lecture et écriture.

Le dernier argument est la fonction de `callback` qui sera exécutée après que l'identifiant ait été créé et que le fichier soit prêt.

Les principaux `flags` disponibles

`'a'` : a pour `appending` c'est-à-dire que le fichier est créé si il n'existe pas.

`'a+'` : a pour `appending` et `+` pour lecture.

`'r'` : ouverture du fichier pour lecture. Une exception est soulevée si le fichier n'existe pas. C'est le `flag` par défaut.

`'r+'` : ouverture du fichier pour lecture et écriture. Une exception est soulevée si le fichier n'existe pas.

`'w'` : ouverture du fichier pour écriture. Le fichier est créé si il n'existe pas ou tronqué si il existe.

`'w+'` : ouverture du fichier pour écriture et lecture. Le fichier est créé si il n'existe pas ou tronqué si il existe.

La méthode `fs.close()`

Cette méthode va en fait supprimer l'identifiant pour le laisser à nouveau disponible.

Cela libère aussi les ressources utilisés pour les opérations sur le fichier.

Nous verrons qu'elle n'est utile que si vous utilisez des méthodes qui créent des `file descriptors`.

Si vous oubliez de `close()` sur les méthodes créant des `fd` vous aurez l'erreur `EMFILE: too many open file` si vous avez ouvert trop de fichiers.