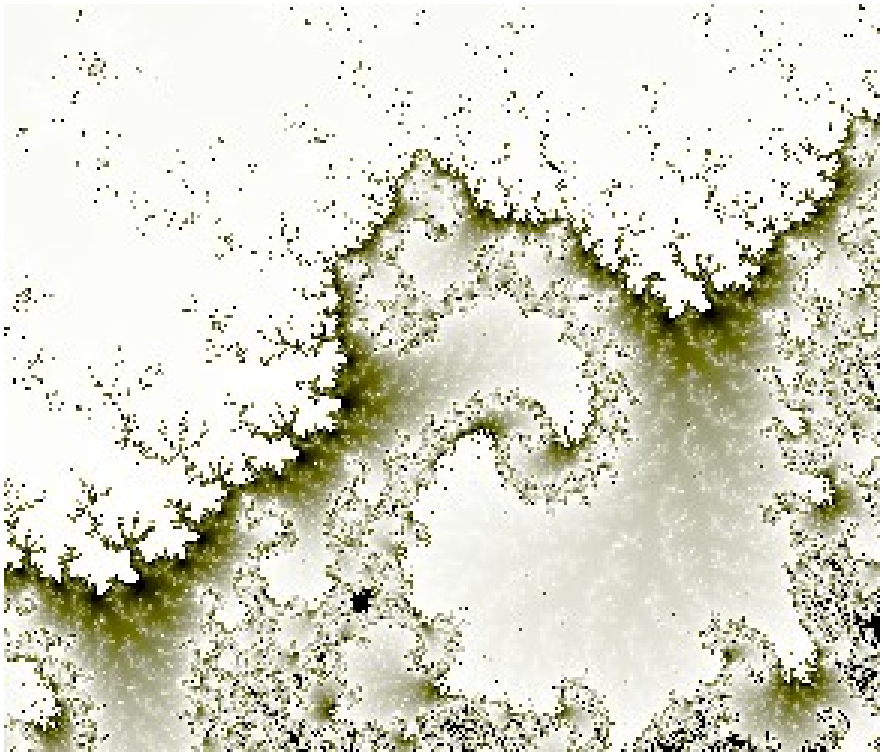
Javascript et les Workers



Version 1.0

Table des matières

Notes de l’auteur :

Je m’appelle Grégory Jarrige.

Je suis développeur professionnel depuis 1991. Après avoir longtemps travaillé sur le développement d’applications de gestion, sur gros systèmes, avec des langages et technos propriétaires, j’ai fait le pari de me former aux technos et langage open source vers 2006. J’ai commencé à développer des applications webs professionnelles à partir de 2007, avant d’en faire mon activité principale à partir de 2010. L’arrivée du HTML5 dans la même période a été pour moi une véritable bénédiction, et surtout un formidable terrain d’expérimentation (avec des API comme Canvas, WebAudio, etc…).

Aujourd’hui, je développe des interfaces utilisateurs en Javascript sur des applications de type intranet, ainsi que des tableaux de bord basés sur des solutions de dataviz. De plus en plus amené à travailler sur des volumes de données importants, aussi bien côté navigateur que côté serveur (sous NodeJS), j’ai éprouvé le besoin de faire le point sur les possibilités du langage Javascript. Car le langage Javascript évolue vite, et si le web regorge d’articles traitant régulièrement des nouveautés du langage, il n’est pas toujours facile de s’y retrouver.

Ce document n’est pas exhaustif, il est forcément partiel et certainement aussi un peu partial. C’est un “work in progress”, réalisé sur mon temps libre, pour mes propres besoins, et j’espère qu’il pourra être utile à d’autres développeurs, qu’ils soient amateurs ou professionnels.

Il est disponible en téléchargement libre sur mon compte Github, dans le dépôt suivant :

https://github.com/gregja/JSCorner

Introduction

Le langage Javascript est un langage basé sur une architecture monothread ce qui signifie qu'il s'appuie sur une file d'attente unique d'exécution. Quand une tâche est exécutée par l'interpréteur Javascript, les autres tâches qui peuvent être lancées simultanément, soit par cette même tâche (par exemple via la fonction setTimeOut), soit par des événements déclenchés par un utilisateur (comme un clic de souris sur un bouton) sont mises en attente dans la file d'attente de travaux. Dès que le thread a terminé la tâche en cours, il prend en charge la tâche suivante, suivant le principe du FIFO (premier entré premier sorti).

Ce mode de fonctionnement asynchrone est efficace et réactif sur la plupart des tâches interactives, qui sont généralement rapides. Mais cela pose des problèmes dans le cas de traitements longs, car dans ce cas l'utilisateur a la désagréable sensation que son navigateur est « gelé ». Cela peut se produire dans le cas d'un calcul gourmand en CPU, ou encore lors du parsing d'un gros fichier JSON (à l'issue d'une requête XMLHTTP), de la construction d'un graphe SVG complexe, etc…

Heureusement, le W3C a ajouté à la norme HTML5 une API permettant de créer des threads secondaires, et donc de paralléliser les tâches : l'API « Web Worker ».

Côté NodeJS, la version 10 a introduit un module « Worker Threads » qui présente beaucoup de similitudes avec l'API « Web Worker » de HTML5 (on peut supposer que ses concepteurs se sont largement inspirés de l'API « Web Worker »).

ATTENTION : il existe plusieurs catégories de workers, côté navigateur. La catégorie que nous allons étudier ici, c'est celle dite des « workers dédiés » (en anglais « dedicated workers »). Pour de plus amples précisions sur les différentes catégories de workers, on pourra se reporter à la documentation suivante :

https://developer.mozilla.org/fr/docs/Web/API/Web_Workers_API

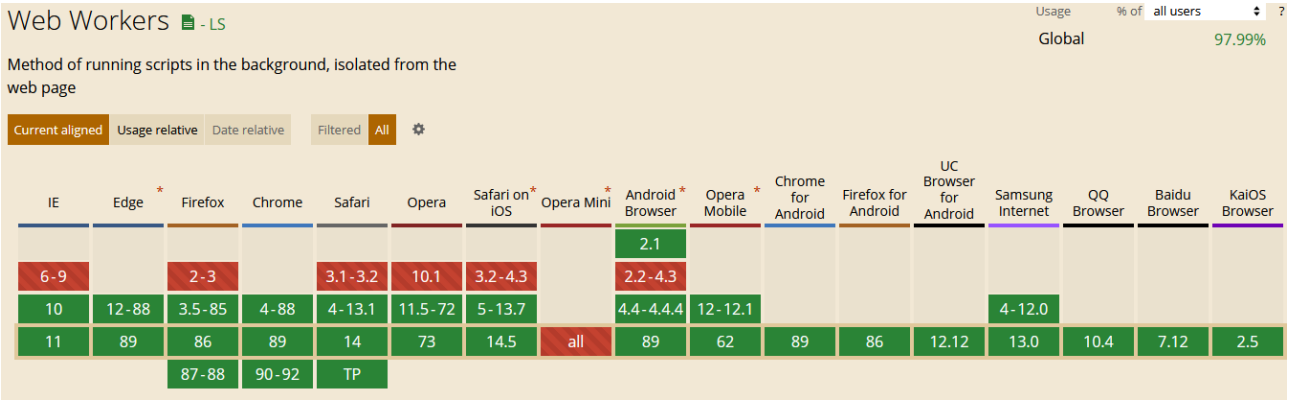
API WebWorker du HTML5

Présentation générale

Il est possible de tester si un navigateur supporte l'API Web Worker via le test suivant :

```
if (window.Worker) {
...
}
```

Le site caniuse.com indique que l'API est supportée par la majorité des navigateurs (cf. copie d'écran ci-dessous prise au 17 mars 2021) :



Pour démarrer un worker dédié s'exécutant dans un thread secondaire, on instancie un objet Worker en lui transmettant le chemin d'accès au script contenant le code du Worker :

```
var monWorker = new Worker('worker.js');
```

L'API Web Worker permet de faire presque tout. On peut par exemple l'utiliser pour lancer des requêtes XMLHTTP destinées à appeler des API, ou à récupérer des fichiers JSON sur un serveur. Ces requêtes XMLHTTP peuvent être lancées via l'objet XMLHttpRequest (ou son équivalent JQuery) ou via la nouvelle API Fetch .

IMPORTANT : l'API Web Worker n'a pas accès au DOM, ce qui signifie qu'on ne peut pas l'utiliser pour mettre à jour directement des portions de page. En revanche on peut l'utiliser pour prémâcher du code HTML ou JSON et le renvoyer au thread principal, qui lui se chargera de la mise à jour du DOM.

A l'intérieur du script "worker.js", on peut embarquer du code provenant de diverses librairies (jQuery ou autre), via la fonction « importScripts », comme dans cet exemple :

```
importScripts('../plugins/alasql.js');
```

A partir de là, la fonction javascript « alasql » (fournie par le script « alasql.js ») est directement accessible dans le script du worker, comme dans cet exemple :

```
let query = `SELECT r.id, r.axeY FROM ? AS r WHERE
NOT( r.min_fin_exec <= ? OR ? <= r.min_deb_exec )
AND r.id != ? and r.axeY = "null" `;
let params = [_dataGroupes, item.startAfter, item.min_fin_exec,
    item.id
];
let conflits = alasql(query, params);
```

On peut bien évidemment embarquer jQuery ou d'autres librairies.

Donc du code écrit initialement pour le mode monothread peut être facilement déporté dans un thread secondaire, en général sans nécessiter de réécriture.

A noter que l'on peut aussi charger un script à partir d'éléments variables, comme dans cet exemple :

```
importScripts(`wkr_loaddatas_tarifs_${data_code}.js`);
```

NB : il peut être intéressant de préfixer les scripts relatifs aux workers, par exemple avec « *wkr_* », pour les distinguer plus facilement des autres scripts.

Echange de messages entre le thread principal et ses workers

Du thread principal vers le worker

Le thread principal n'a pas accès aux fonctions Javascript déclarées au sein du worker, mais il peut lui transmettre des messages, via la fonction « postMessage », comme dans l'exemple suivant :

```
var monWorker = new Worker('worker.js');
monWorker.postMessage([premierNombre.value, deuxiemeNombre.value]);
```

Une bonne astuce consiste à réserver le premier paramètre à un code action permettant d'indiquer au worker ce qu'il doit faire, les paramètres suivants étant réservés à la transmission des données « métiers ». La plupart des exemples trouvés sur internet montrent la transmission d'un tableau (comme l'exemple ci-dessus), mais on peut aussi transmettre un objet, comme dans l'exemple ci-dessous :

```
worker.postMessage({action: 'fetch', id: fetch_id, context: context})
```

Attention : tout objet transmis à la fonction « postMessage » est transmis par valeur. Il faut donc être prudent quant au volume d'informations que l'on transmet à la fonction « postMessage ». C'est un problème que l'on retrouvera avec l'API Worker Threads de NodeJS.

A l'intérieur du worker

A l'intérieur du worker, on récupère en entrée un paramètre unique contenant une propriété « data ». Cette propriété « data » contient la liste des paramètres transmis à la fonction « postMessage » par le thread principal. La propriété « data » sera donc soit un tableau, soit un objet, selon ce qui a été transmis à « postMessage ». Pour récupérer le contenu de la propriété « data », le code est le suivant :

```
self.onmessage = function (param) {
    "use strict";
    var data = param.data;
    if (data.action == 'init') {
// ...
    }
    if (data.action == 'fetch') {
// ...
    }
}
```

Du worker vers le thread principal

Le worker renvoie des réponses au thread principal en utilisant la syntaxe suivante :

```
self.postMessage({status: 'OK', message:'tout va bien'})
```

La présence systématique d'une propriété « status » dans le message renvoyé par le worker constitue une bonne pratique car elle permet au thread principal de savoir instantanément si le worker a bien travaillé ou pas.

Traitement dans le thread principal des messages émis par le worker

Dans l'exemple de la page précédente, nous avons instancié un worker avec la syntaxe suivante :

```
var monWorker = new Worker('worker.js');
```

Pour réceptionner – au niveau du thread principal - les messages renvoyés par le worker, il faut ajouter un écouteur d'événement sur le worker, via le code suivant :

```
monWorker.onmessage = function(e) {
    console.log('Message reçu depuis le worker => ', e.data);
}
```

Organisation du code à l'intérieur du worker

Pour éviter que le code à l'intérieur des workers ne devienne un gros bazard, on peut structurer son code en deux parties :

1. Un objet (appelé ici WorkerLab) pour le stockage des données et des fonctions métier
2. Une méthode distincte (anonyme) associée à la propriété « onmessage » de l'objet « self », pour gérer les échanges « bas niveau » avec le thread principal

```
var WorkerLab = (function () {
    "use strict";
    //--- constantes privées
    const path_API = 'http://xxx.yyy.zzz';
    // url d'API utilisée par l'API Fetch

    //--- variables privées
    var datas = {}; // exemples d'objet pour le stockage de données "métier"

    //--- méthodes privées
    function _transformData(data) {
        // méthode (privée) de transformation de la variable "data"
        // selon les règles "métier" du worker
        return xxx; // envoi du résultat de la transformation de "data"
    }

    //--- méthodes publiques
    function init(id, context) {
        // stockage des données "métier" reçues à l'initialisation
        datas.id = id;
        datas.context = context;
    }

    function fetch(worker) {
        // Requête de type GET avec l'API Fetch qui fonctionne en mode
        // asynchrone (via l'API Promise sous-jacente).
        // Le mode asynchrone oblige à placer les appels à "postMessage"
        // tel qu'indiqué dans l'exemple ci-dessous
        fetch(this.path_API)
            .then(res => res.json())
            .then(data => {
                // transformation des données avant envoi au thread principal
                let tmp = this._transformData(data);
```

```
                // envoi du résultat au thread principal si tout est OK
                worker.postMessage({
                    status: "OK",
                    data: tmp
                });
            }).catch(err => {
                // prévenir le thread principal du fait que le Fetch a échoué
                worker.postMessage({
                    status: "BAD",
                    error: err.message
                });
            });
    }

    // Déclaration des méthodes et propriétés publiques
    return {
        init: init,
        fetch: fetch
    };
})();
self.onmessage = function fetchworker(param) {
    "use strict";
    var data = param.data;
    var worker = this; // création d'un pointeur sur l'instance du worker
    // pour transmission aux méthodes de WorkerLab si besoin
    if (data.action == 'init') {
        // appel de la fonction d'initialisation de l'objet "métier"
        WorkerLab.init();
        // envoi d'un message au thread principal pour lui indiquer
        // que l'initialisation s'est bien passée
        self.postMessage({status: 'OK'});
    }
    if (data.action == 'fetch') {
        // on transmet l'instance du worker à la méthode "fetch" pour que
        // cette dernière puisse exécuter des "postMessage" directement
        WorkerLab.fetch(worker);
    }
}
```

Arrêt du worker

L'arrêt d'un worker se fait au niveau du thread principal, en utilisant l'instruction suivante :

```
monWorker.terminate();
```

Attention : certaines documentations (non à jour) laissent entendre qu'il serait possible à un worker de s'arrêter lui-même via la méthode « close ». Il aurait donc été possible pour un worker de s'arrêter de lui-même sous certains conditions, comme par exemple une demande d'arrêt émanant du thread principal. Cela aurait donné ceci :

```
if (data.action == 'exit') {
    self.close(); // fermeture définitive du worker par lui-même
                  // (si demandé par le thread principal)
}
```

En fait cela a été possible pendant quelques temps, mais cette possibilité est maintenant dépréciée, il est donc vivement recommandé de ne pas l'employer. Pour de plus amples précisions sur ce sujet :

https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Functions_and_classes_available_to_workers

Création d'un worker à la volée

Nous avons vu comment instancier un worker à partir d'un code Javascript stocké dans un fichier Javascript externe.

Mais il est possible de créer un worker à la volée, c'est-à-dire en construisant son code dynamiquement.

Attention : certains tutos non à jour disponibles sur internet présentent des exemples utilisant l'objet Javascript BlobBuilder pour la construction dynamique de code Javascipt. Cet objet BlobBuilder est déprécié et les exemples en question ne fonctionnent plus.

Vous trouverez ci-dessous deux variantes qui fonctionnent toutes deux.

Variante n° 1 : creation d'un worker à partir d'une chaîne de caractère

```
// chaine de caractère contenant le code d'une fonction JS
var dyn_code = `onmessage = function(e) {
console.log('Message reçu par le worker : ' +
e.data);
postMessage('Salut "thread principal", ' +
' ça va bien merci');
}`;
var blob = new Blob([dyn_code]);
// Création d'une reference de type "blob URL"
var blobURL = window.URL.createObjectURL(blob);
// creation du Worker
var myworker = new Worker(blobURL);
myworker.onmessage = function(e) {
    console.log('Message reçu par le thread principal :' + e.data);
};
myworker.postMessage(`salut "worker enfant", ça va comme tu veux ?`);
```

Variante n° 2 : création d'un worker à partir d'une fonction anonyme

```
var code_for_dyn_worker = function(){
    self.onmessage = function(e) {
        console.log('Message reçu par le worker : '+e.data);
        postMessage('Salut "thread principal", ça va bien merci');
    }
};
var BuildWorker = function(foo){
    var str = foo.toString()
        .match(/^\s*function\s*\(\s*\)\s*\{(([\s\S](?!\}$))*[\s\S])/)[1];
    return new Worker(window.URL.createObjectURL(
        new Blob([str],{type:'text/javascript'})
    ));
};
var myworker = BuildWorker(code_for_dyn_worker);
myworker.onmessage = function(e) {
    console.log('Message reçu par le thread principal :' + e.data);
};
myworker.postMessage(`salut "worker enfant", ça va comme tu veux ?`);
```

Dans cette seconde variante, on transforme provisoirement en chaîne de caractères la fonction anonyme contenue dans la variable « code_for_dyn_worker ». C'est la fonction BuildWorker qui fait « le gros du boulot » (solution trouvée sur Stackoverflow).

NodeJS et les Worker Threads

Présentation générale

Le module natif « Worker Threads » est apparu sur la V10 de NodeJS.

Lien vers la documentation officielle (qui n'est pas d'une grande clarté) :

https://nodejs.org/dist/latest-v10.x/docs/api/worker_threads.html

En 2019, ce module était expérimental et n'était pas activé par défaut. Pour l'activer, il fallait ajouter le paramètre optionnel suivant :

```
node --experimental-worker monscript.js
```

En 2021, avec notamment la version 14 de Node.js, le module est désormais stable et est activé par défaut.

Premier exemple en mode « pas à pas »

A l'intérieur du script "monscript.js", nous devons importer l'API « worker threads » via l'instruction suivante :

```
const { Worker, isMainThread, parentPort, workerData } = require('worker_threads');
```

Nous devons ensuite instancier un worker. Nous avons pour cela deux possibilités :

- Soit nous instancions le worker en lui indiquant d'utiliser comme code de base le script courant avec la constante « __filename » :

```
let worker = new Worker(__filename, { workerData: { value: 123456789 }});
```

- Soit nous instancions le worker en utilisant pour code de base un autre script :

```
let worker = new Worker('./monworker.js', { workerData: { value: 123456789 }});
```

Pour que le thread principal puisse dialoguer avec un thread enfant (le worker), nous devons mettre en place les écouteurs d'événement, que voici :

```
worker.on('error', (err) => { throw err; });
worker.on('exit', () => {
    console.log('Thread exiting'); // détection de l'arrêt du worker
})
worker.on('message', (msg) => {
    console.log(msg); // traitement des messages émis par le worker
});
```

Si nous avons instancié notre worker avec la constante « __filename », cela signifie que le même code s'exécute à la fois dans le thread principal et dans le thread « enfant ».

Si certaines parties de code peuvent être utilisées indifféremment dans les deux threads, nous avons besoin de savoir distinguer les parties de code qui doivent s'exécuter uniquement dans l'un et l'autre thread.

Cette distinction se fait via le booléen « isMainThread », comme ceci :

```
if (isMainThread) {
    // partie 1 : code qui s'exécute dans le thread principal uniquement
    let value_to_calc = 123456789;
    let worker = new Worker(__filename, { workerData: { value: value_to_calc }});
    worker.on('error', (err) => { throw err; });
    worker.on('exit', () => {
        console.log('Thread exiting');
    })
    worker.on('message', (msg) => {
        console.log(msg);
    });
} else {
    // partie 2 : code qui s'exécute dans le thread enfant uniquement
    let result = mafonction(workerData.value);
    parentPort.postMessage(result);
}
```

Dans l'exemple qui précède, le code qui s'exécute dans la partie 2 est spécifique au worker enfant. C'est l'exemple que l'on rencontre le plus souvent dans les tutoriaux disponibles sur internet, mais il est incomplet, car dans ce mode de fonctionnement, le worker ne s'exécute qu'une seule fois, puis il se « ferme » automatiquement, ce qui a pour effet de déclencher l'événement « exit » au niveau du thread principal. Cela peut être pratique dans certains cas particuliers, mais dans la « vraie vie », on a généralement besoin de réutiliser un même worker plusieurs fois.

Pour empêcher que le worker ne s'arrête automatiquement après le premier appel, il faut modifier le code de la façon suivante (les parties modifiées sont en gras) :

```
if (isMainThread) {
    let value_to_calc = 20;
    let worker = new Worker(__filename,{workerData:{name:"fibo",action:"wait"}});
    worker.on('error', (err) => { throw err; });
    worker.on('exit', () => {
        console.log('Thread exiting');
    })
    worker.on('message', (msg) => {
        console.log(msg);
    });
    worker.postMessage({action: "calc", input:value_to_calc});
} else {
    parentPort.on("message", message => {
        if (message.action === "exit") {
            parentPort.close();
        } else {
            if (message.action === "calc") {
                parentPort.postMessage(mafonction(message.input));
            }
        }
    });
    console.log('parameter received by worker '+workerData.name+' on init : ',
        workerData.action);
}
```

Dans l'exemple qui précède, lors de l'instanciation du worker, nous avons transmis un code action « wait ». C'est une information facultative, elle n'est demandée par l'API, mais elle est pratique pour suivre le traitement en mode debug et alimenter la log Javascript.

Le worker ne fait donc aucune action au moment de l'instanciation :

```
let worker = new Worker(__filename,{workerData:{name:"fibo",action:"wait"}});
```

… car ce même worker "sait" qu'il doit attendre l'arrivée de messages en provenance de son parent, grâce à ce code (déclaré dans la partie 2) :

```
parentPort.on("message", message => {
…
});
```

C'est un peu plus loin dans la partie 1, que le thread principal va demander au thread enfant de se mettre au travail, via la ligne suivante :

```
worker.postMessage({action: "calc", input:value_to_calc});
```

Grâce au code ci-dessous déclaré dans la partie 2, et grâce à la présence du code « action » dans le message reçu, le worker « sait » s'il doit s'arrêter ou calculer une nouvelle donnée :

```
parentPort.on("message", message => {
    if (message.action === "exit") {
        parentPort.close();
    } else {
        if (message.action === "calc") {
            parentPort.postMessage(mafonction(message.input));
        }
    }
});
```

Le chapitre suivant présente un exemple d'implémentation complet, avec une fonction métier implémentant le calcul d'un nombre de Fibonacci. Dans cet exemple, le worker enfant est appelé 5 fois avec à chaque fois une nouvelle valeur à calculer, puis il reçoit un dernier message lui demandant de s'arrêter.

Calcul nombre de Fibonacci avec un seul worker

```
'use strict';
const { Worker, isMainThread, parentPort, workerData } = require('worker_threads');
function fibonacci(n){
    let arr = [0, 1];
    for (let i = 2; i < n + 1; i++){
        arr.push(arr[i - 2] + arr[i -1])
    }
    return {input:n, output: arr[n]};
}
if (isMainThread) {
    var values_to_calc = [20, 2, 50, 160, 33];
    var nb_values_to_calc = values_to_calc.length;
    let worker = new Worker(__filename,{workerData:{name:"fibo",action:"wait"}});
    worker.on('error', (err) => { throw err; });
    worker.on('exit', () => {
        console.log('Thread exiting');
    })
    worker.on('message', (msg) => {
        console.log(msg);
        nb_values_to_calc--;
        if (nb_values_to_calc == 0) {
            worker.postMessage({action: "exit"});
        }
    });
    values_to_calc.forEach(value_to_calc => {
        worker.postMessage({action: "calc", input:value_to_calc});
    });
} else {
    parentPort.on("message", message => {
        if (message.action === "exit") {
            parentPort.close();
        } else {
            if (message.action === "calc") {
                parentPort.postMessage(fibonacci(message.input));
            }
        }
    });
    console.log('parameter received by worker '+workerData.name+' on init : ',
        workerData.action);
}
```

Le résutat obtenu en sortie est le suivant :

```
$ node --experimental-worker fibonacci_vxx.js
parameter received by worker fibo on init : wait
{ input: 20, output: 6765 }
{ input: 2, output: 1 }
{ input: 50, output: 12586269025 }
{ input: 160, output: 1.2261325953941887e+33 }
{ input: 33, output: 3524578 }
Thread exiting
```

Vous noterez dans cet exemple qu'il faut bien faire attention à la manière dont l'arrêt du worker a été implémenté :

```
worker.on('message', (msg) => {
    console.log(msg);
    nb_values_to_calc--;
    if (nb_values_to_calc == 0) {
        worker.postMessage({action: "exit"});
    }
});
```

Explication : à chaque nouvelle réponse renvoyée par le worker enfant, on décrémente un compteur contenant le nombre de valeurs à calculer. Quand ce compteur arrive à zéro, on appelle le worker une dernière fois en lui demandant de s'arrêter.

Calcul nombre de Fibonacci avec autant de workers que de valeurs à calculer

Dans cette variante, on démarre autant de workers qu'il y a de valeurs à calculer. Du coup chaque worker s'arrête automatiquement après sa première utilisation, et on affiche les résultats obtenus quand il n'y a plus aucun worker actif.

C'est une solution intéressante, mais attention, il semble que l'API Worker Threads pose des problèmes quand on dépasse 8 ou 9 workers actifs en même temps. De plus, le principe consistant à démarrer autant de workers que de données à calculer est peut être intéressant pour un « one shot », mais certainement pas pour développer une solution robuste et scalable (la solution présentée au chapitre suivant est certainement plus satisfaisante).

```
'use strict';
const { Worker, isMainThread, parentPort, workerData } = require('worker_threads');
const fibonacci = function(n){
    let arr = [0, 1];
    for (let i = 2; i < n + 1; i++){
        arr.push(arr[i - 2] + arr[i -1])
    }
    let result = {input:n, output: arr[n]};
    return [result];
}
if (isMainThread) {
    let results = [];
    var values_to_calc = [20, 2, 50, 160, 33];
    const threads = new Set();
    console.log(`Running with ${values_to_calc.length} threads...`);
    for (let i = 0, imax=values_to_calc.length; i < imax; i++) {
        console.log(`Thread ${i} runs with value : ${values_to_calc[i]}`);
        threads.add(new Worker(__filename,{workerData:{value:values_to_calc[i]}}));
    }
    for (let worker of threads) {
        worker.on('error', (err) => { throw err; });
        worker.on('exit', () => {
            threads.delete(worker);
            console.log(`Thread exiting, ${threads.size} running...`);
            if (threads.size === 0) {
                console.log('finish !!!')
                for (let i=0, imax=results.length; i<imax; i++) {
                    let result = results[i][0];
                    console.log(result);
                }
            }
        })
        worker.on('message', (msg) => {
            results.push(msg);
        });
    }
} else {
    parentPort.postMessage(fibonacci(workerData.value));
}
```

Calcul nombre de Fibonacci avec un pool de workers

Dans la variante présentée dans ce chapitre, on définit un pool de 4 workers que l'on instancie et que l'on met en attente. On définit également un certain nombre de valeurs à calculer, et on lance le pool de workers à l'assaut de ce tableau de valeurs. Dès qu'un worker a fini de travailler, il regarde dans le tableau des valeurs s'il reste des valeurs à calculer. S'il ne reste plus aucune valeur à calculer, un message d'arrêt est envoyé à l'ensemble des workers et le résultat de leurs travaux respectifs est affiché.

Pour implémenter le pool de workers, on a utilisé l'objet JS standard « Map » qui est très pratique pour gérer des séries d'objets :

```
'use strict';
const { Worker, isMainThread, parentPort, workerData } = require('worker_threads');
if (isMainThread) {
    let values_to_calc = [20, 2, 50, 160, 33, 45, 55];
    // array to store the fibonacci results
    let results = [];
    // name of the workers used by the pool
    let pool = ['worker1', 'worker2', 'worker3', 'worker4'];
    function runService(workerData) {
        const threads = new Map();
        console.log(`Calc ${values_to_calc.length} values dispatched between $
{pool.length} threads...`);
        // Start the pool of workers
        for (let i = 0, imax=pool.length; i < imax; i++) {
            threads.set(pool[i], new Worker(__filename,
                { workerData: { name:pool[i], data: "wait" }}));
        }
        // when all numbers are calculated, we can close all workers of the pool
        function closeWorkers() {
            for (let [key, worker] of threads.entries()) {
                worker.postMessage({ data: "exit" });
            }
        }
        // attach the calculus of a fibonacci series to a worker transmitted
        function attachJob(worker) {
            if(values_to_calc.length > 0) {
                worker.postMessage({ data: values_to_calc.pop() });
            }
        }
       // create a new worker
        function addOneWorker(key, worker) {
            worker.on('error', (err) => { throw err; });
            worker.on('exit', () => {
                threads.delete(key);
                console.log(`Thread ${key} exiting, ${threads.size} running...`);
                if (threads.size === 0) {
                    console.log('finish !!! Display the results please ;)');
                    for (let i=0, imax=results.length; i<imax; i++) {
                        let result = results[i][0];
                        console.log(result);
                    }
                }
            })
            worker.on('message', (msg) => {
                results.push(msg);
```

```
                if(values_to_calc.length > 0) {
                    attachJob(worker);
                } else {
                    // good job guys, you can close the stores ;)
                    closeWorkers();
                }
            });
        }
        for (let [key, worker] of threads.entries()) {
            addOneWorker(key, worker);
            attachJob(worker);
        }
    }
    async function run() {
        runService("let's begin");
    }
    run().catch(err => console.error(err));
} else {
    const fibonacci = function(n){
        let arr = [0, 1];
        for (let i = 2; i < n + 1; i++){
            arr.push(arr[i - 2] + arr[i -1])
        }
        let result = {input:n, output: arr[n]};
        return [result];
    }
    parentPort.on("message", message => {
        if (message.data === "exit") {
            parentPort.close();
        } else {
            parentPort.postMessage(fibonacci(message.data));
        }
    });
    console.log('parameter received by worker '+workerData.name+' on init : ',
        workerData.data);
}
```

Contraintes de l'API Worker Threads

Si l'API Worker Threads offre des possibilités intéressantes, elle pose aussi certaines contraintes qui en limitent l'usage.

- il est possible de partager un modèle de données entre workers, mais la structure de ce modèle de données est assez limitée car il s'agit d'un objet de type "ArrayBuffer".

- il est possible de transmettre des données complexes entre le thread principal et ses enfants via le mécanisme standard de transmission de message de l'API Worker Threads. Mais la transmission des données se fait par valeur (copie intégrale des valeurs) et non par adresse, et ce même pour les objets.

- une alternative pour le partage de données entre thread principal et workers consisterait à utiliser une base de données – par exemple MongoDB - comme mémoire partagée entre les différents threads (le principal comme ses enfants). Le chapitre suivant présente un exemple d'utilisation de MongoDB dans ce contexte.

MongoDB comme mémoire partagée pour les threads

L'exemple suivant est en deux parties. Le script principal intancie deux workers ayant des rôles distincts :

- Le premier worker est appelé 2 fois. A chaque appel, il insère des valeurs aléatoires dans une collection Mongo
- Le second worker est lancé 3 secondes plus tard, et va lire dans la collection Mongo les données générées par le premier worker

C'est un exemple basique, mais qui a le mérite de montrer la faisabilité de l'utilisation de Mongo au sein de worker threads.

Code source du thread principal :

```
const { Worker, isMainThread, parentPort, workerData } = require('worker_threads');
function getRandomInt(max) {
    return Math.floor(Math.random() * Math.floor(max));
}
var script_worker = './mongoworker_worker.js';
var threads = [];
var wkr1 = new Worker(script_worker, { workerData: { action: "wait" }});
threads.push(wkr1);
var wkr2 = new Worker(script_worker, { workerData: { action: "wait" }});
threads.push(wkr2);
threads.forEach((wkr, idx) => {
    let val = (idx+21)*10*(getRandomInt(5)+1); // fake data
    wkr.postMessage({ data: [{a : val}], action:"insert" });
    val = (idx+21)*10*(getRandomInt(5)+1); // fake data
    wkr.postMessage({ data: [{a : val}], action:"insert" });
});
setTimeout(function(){
    wkr1.postMessage({ data: [], action:"read" });
}, 3000);
```

Code source des workers enfants :

```
const { Worker, isMainThread, parentPort, workerData } = require('worker_threads');
const MongoClient = require('mongodb').MongoClient;
// Connection URL
const url = 'mongodb://localhost:27017';
// Database Name
const dbName = 'myproject';
function mongoWorker (action, newDatas) {
    MongoClient.connect(url, { useNewUrlParser: true }, function(err, client) {
        console.log("Connected successfully to server");
        const db = client.db(dbName);
        if (action == 'insert') {
            insertDocuments(db, newDatas, function() {
                client.close();
            });
        }
        /*
            insertDocuments(db, newDatas, function() {
                findDocuments(db, function() {
                    client.close();
                });
            });
        */
        if (action == 'read') {
            findDocuments(db, function() {
                client.close();
            });
        }
    });
}
const insertDocuments = function(db, datas, callback) {
    // Get the documents collection
    const collection = db.collection('documents');
    // Insert some documents
    collection.insertMany(datas, function(err, result) {
        let count = datas.length;
        /* assert.equal(err, null);
        assert.equal(count, result.result.n);
        assert.equal(count, result.ops.length);*/
        console.log(`Inserted ${count} documents into the collection`);
        callback(result);
    });
}
const findDocuments = function(db, callback) {
    // Get the documents collection
    const collection = db.collection('documents');
    // Find some documents
    collection.find({}).toArray(function(err, docs) {
        // assert.equal(err, null);
        console.log("Found the following records");
        console.log(docs)
        callback(docs);
    });
}
```

```
if (!isMainThread) {
    parentPort.on("message", message => {
        if (message.action === "exit") {
            parentPort.close();
        } else {
            if (message.action != "wait") {
                parentPort.postMessage(mongoWorker(message.action, message.data));
            }
        }
    });
    console.log('parameter received by worker on init : ', workerData.data);
}
```