Editions ENI

Node.js

Exploitez la puissance de JavaScript côté serveur

Collection
Expert IT

Extrait

Chapitre 7
Promesses

1. Introduction

Ce chapitre est dédié aux promesses. Mais avant d'explorer ce concept, il est pertinent de comprendre pourquoi l'utiliser.

S'ensuivra une analyse détaillée d'une promesse : ses états, l'approche semblabe à du code synchrone et la gestion des erreurs. Puis, nous verrons comment créer une promesse de différentes façons, ce qui permettera de bien appréhender le concept. Enfin, nous explorerons l'intégration des promesses dans Node et dans les générateurs.

2. Callbacks vs promesses

L'approche traditionnelle avec Node est d'utiliser des callbacks. Ainsi, on ne bloque pas le serveur en attendant d'exécuter une tâche : celle-ci est lancée, et une fois terminée, elle appelle une fonction.

Cependant, les callbacks souffrent de plusieurs maux. Et le premier est le problème de l'imbrication : effectivement, en cas d'appels successifs à de nombreuses callbacks, le code devient rapidement illisible, et pire, dur à maintenir.

De surcroît, il faut gérer les erreurs à la main : d'ailleurs, si l'on oublie d'en traiter une, on fait face à de gros problèmes (une erreur non traitée est difficile à identifier pendant l'exécution du programme). Ce n'est pas tout : les exceptions donnent aussi quelques sueurs froides. Par exemple, il suffit d'en lancer une au sein d'une callback pour obtenir un système instable car personne ne peut la rattraper (`try {} catch {}`). En outre, Node ne supporte pas les exceptions non gérées. Sans oublier le fait qu'il est impossible de récupérer la pile d'erreurs complète ! Vraiment gênant pour le débogage.

Autre subtilité, il est nécessaire de faire très attention à la manière dont on appelle une callback. Synchrone ou asynchrone, il faut choisir, car si l'on mélange les paradigmes, il est de coutume de dire qu'on libère « Zalgo » (monstre de l'imaginaire populaire, sorte d'abomination qui rend fou). Et ce genre de comportement rend effectivement fou : impossible de prédire le comportement de la callback ! Et au final, de quoi s'arracher les cheveux pour trouver l'origine du problème.

Enfin, pour clore cette longue liste de difficultés, une callback doit par définition être appelée une fois au maximum. Si ce point peut paraître étrange, sachez qu'un appel multiple par erreur peut arriver dans une fonction un peu longue ou complexe, avec par exemple un abus de copier/coller. Toujours est-il que rien ne protège de cette erreur.

Mais heureusement, les promesses sont là pour remédier à ces problèmes. Et c'est exactement pour cela qu'elles bénéficient d'un chapitre complet.

Un exemple est bien entendu beaucoup plus parlant. Voici le code de la section Programmation asynchrone du chapitre Concepts, réécrit en utilisant les promesses et l'API de Bluebird (utilisée tout au long du chapitre).

```
var Bluebird = require('bluebird');

var got = Bluebird.promisify(require('got'));
var writeFile = Bluebird.promisify(require('fs').writeFile);

var files = [
    {
        name: 'foo.html',
        url: 'http://example.org/foo.html',
    },
    {
```

```
        name: 'bar.html',
        url: 'http://example.org/bar.html',
    },
];

Bluebird

    // Télécharge en parallèle tous les fichiers de `files`.
    .map(files, function (file) {
        return got(file.url);
    })

    // Puis les enregistre sur le disque (toujours en parallèle).
    .map(function (result, i) {
        // got() retourne deux résultats que Bluebird.promisify()
        // transforme en tableau.
        var content = result[0];

        return writeFile(files[i].name, content);
    })

    .then(function () {
        console.log('Tout s\'est bien passé');
    })
    .catch(function (error) {
        console.error(error);
    })
;
```

En premier, il ne faut pas oublier d'inclure la bibliothèque `bluebird` via `require()` (il sera expliqué ce qu'est Bluebird plus loin). La différence entre le code du chapitre Concepts et celui-ci commence juste après : les deux fonctions requises (`got` et `writeFile`) sont « promessifiées » (voir la section Création d'une promesse - À partir d'une fonction Node) avec la fonction `promesify()` que vous allez observer tout au long de ce chapitre.

Les deux fichiers sont toujours foo.html et bar.html, dans la variable `files`.

Ensuite, le premier appel à `Bluebird.map()` applique la fonction `got` à tous les éléments du tableau, autrement dit : on télécharge en parallèle tous les fichiers présents dans `files`. L'enregistrement sur le disque se passe aussi via un appel à `map()` qui met les résultats de la lecture des fichiers dans un tableau. Ces résultats sont écrits via `writeFile()`, et tout à la fin, à la dernière étape, au niveau de `then()`, est affiché un message indiquant si tout s'est bien déroulé (ou pas).

La gestion des erreurs se situe à un seul endroit, au niveau de `catch()` ! Comme on peut le constater, l'écriture d'un tel code est fortement simplifiée comparativement à la version callbacks. Et ce n'est que le début : il est temps de comprendre maintenant le concept de promesse.

3. Notion de promesse

La promesse est un paradigme de programmation asynchrone qui résout avec style beaucoup de problèmes que l'on trouve en utilisant le passage de continuations (la convention dans Node).

Et concrètement ? Une promesse est un objet représentant une valeur qui sera disponible dans le futur.

Par exemple, avec `readFile()` qui est asynchrone, le contenu du fichier foo.txt ne sera disponible que plus tard :

```
var promise = readFile('foo.txt');
```

Il est possible d'accéder à la valeur d'une promesse en enregistrant une fonction callback avec la méthode `.then(callback)` qui sera appelée quand cette valeur sera disponible. Voici ce que cela donne pour afficher le résultat de `promise` :

```
promise.then(function (content) {
console.log('le contenu de foo.txt est', content);
});
```

Une promesse peut être rejetée, par exemple si la lecture du fichier a échoué. L'erreur associée à ce rejet peut être récupérée de la même façon qu'est récupérée la valeur, mais avec la méthode `.catch(callback)` :

```
promise.catch(function (error) {
  console.error('la lecture a échoué car', error);
});
```

Les promesses sont tellement pratiques qu'elles seront incluses par défaut dans la prochaine version de JavaScript (ECMAScript 6) avec l'objet `Promise` et qu'elles sont utilisées dans la plupart des nouvelles API de HTML5.

Outre l'implémentation officielle, il existe un grand nombre de bibliothèques implémentant les promesses tout en respectant la spécification standard A+, ce qui leur permet d'être complètement intercompatibles.

Remarque

A+ est un standard open source pour les promesses en JavaScript. Il explicite la terminologie commune et le cahier des charges pour quiconque souhaiterait faire une bibliothèque de promesses. Pour plus de détails sur ce qu'est A+, nous vous invitons à consulter cette URL : https://promisesaplus.com/.

Node ne fournissant pas pour le moment d'implémentation native, il est nécessaire d'utiliser l'une de ces bibliothèques. Historiquement la plus utilisée fut Q (disponible à cette adresse : https://github.com/kriskowal/q), cependant elle est de plus en plus remplacée par Bluebird. Le dépôt officiel et la documentation sont accessibles via cette URL : https://github.com/petkaantonov/bluebird. Non contente d'être l'une des plus performantes (sinon la plus performante), elle propose aussi un nombre impressionnant de méthodes très pratiques permettant de gagner considérablement en efficacité lors du développement avec des promesses.

Dans les exemples de ce chapitre, il est fait référence à `Promise` lorsque seules des méthodes standards et compatibles avec l'implémentation officielle sont utilisées (compatibles avec ES6 et la majorité des implémentations), et à Bluebird lorsque nous utilisons certaines de ses méthodes spécifiques.

3.1 États d'une promesse

Une promesse est dans un état, et un seul. Les états possibles sont :

– en attente (*pending*), quand l'opération asynchrone n'est pas terminée.
– remplie (*fulfilled*), quand l'opération est terminée avec succès.
– rejetée (*rejected*), quand l'opération a échoué.

Une fois que la promesse est remplie ou rejetée, elle ne change plus jamais.

3.2 Similarité avec du code synchrone

Un des objectifs des promesses est de rendre le code asynchrone le plus facile à écrire pour un développeur. L'idée est donc qu'il se rapproche le plus possible du code synchrone.

Voici un code synchrone qui lit un fichier JSON, décode son contenu puis, après l'avoir modifié, le réécrit :

```
try {
    var config = JSON.parse(fs.readFileSync('config.json'));

    config.foo = 'bar';

    fs.writeFile('config.json', JSON.stringify(config));
} catch (error) {
    console.error('Erreur :', error);
}
```

Et voici une version asynchrone basée sur les promesses :

```
var readFile = Bluebird.promisify(fs.readFile);
var writeFile = Bluebird.promisify(fs.writeFile);

readFile('config.json').then(JSON.parse).then(function (config) {
    config.foo = 'bar';

    return writeFile('config.json', JSON.stringify(config));
}).catch(function (error) {
    console.error('Erreur :', error);
});
```

Editions ENI

AngularJS

Développez aujourd'hui les applications web de demain

Collection
Expert IT

Extrait

Chapitre 5
Promises et requêtage HTTP

1. Introduction

Dans un site web classique, le rôle du serveur est de fournir des pages que le navigateur affichera. Dans une SPA, son rôle est différent. Puisque toute la gestion de la navigation, de l'affichage des données et des pages est gérée côté client, en JavaScript, le serveur se libère de ces responsabilités et devient une API Web. Son rôle est de renvoyer des données, généralement au format JSON.

Les notions de promises, permettant de chaîner des actions asynchrones, et les mécanismes de requêtage HTTP fournis par AngularJS seront évoqués dans ce chapitre.

2. Promise, la fin des callbacks

JavaScript est un langage mono-threadé par design et se base sur un modèle événementiel non bloquant. Cela implique que ce langage utilise beaucoup le mécanisme d'actions asynchrones et s'appuie sur le principe de callback pour gérer cet asynchronisme.

Une fonction non asynchrone renvoie une valeur ou un objet.

```
var result = addition(4, 8)
```

La méthode `addition` est synchrone et renvoie le résultat de l'addition entre ses deux paramètres. La variable `result` a alors pour valeur 12.

Une fonction JavaScript asynchrone ne renvoie pas de résultat mais va prendre en paramètre une fonction de callback qui sera appelée lorsque le traitement sera terminé. Cette fonction de callback va elle-même prendre en paramètre les résultats du traitement.

```
step1(function(value1) {
    // Traitement lorsque l'exécution de la fonction est terminée
});
```

La méthode `step1` est asynchrone et prend en paramètre une fonction callback. Cette fonction est appelée lorsque l'exécution de l'action asynchrone est terminée, et prend en paramètre la valeur résultante de l'action.

Bien que ce mécanisme fonctionne, il devient difficile à utiliser lorsqu'il est nécessaire de manipuler plusieurs fonctions asynchrones, par exemple pour effectuer des appels chaînés ou parallèles, ou bien lorsqu'il est souhaitable de mettre en place une gestion des erreurs globales à plusieurs appels de fonctions asynchrones.

```
step1(function (value1) {
    step2(value1, function(value2) {
        step3(value2, function(value3) {
            step4(value3, function(value4) {
                // Traitement lorsque l'exécution des quatre
                // fonctions asynchrones est terminée
            });
        });
    });
});
```

Le code précédent chaîne l'exécution de plusieurs méthodes asynchrones. La méthode `step2` est appelée dans le callback de la méthode `step1`, la méthode `step3` dans le callback de la méthode `step2` et la méthode `step4` dans le callback de la méthode `step3`.

L'imbrication de callbacks, comme vu dans l'exemple précédent, est une syntaxe qui est appelée Pyramid of Doom et qui est une syntaxe rencontrée fréquemment lorsqu'une application JavaScript utilise beaucoup d'asynchronisme. Le problème de cette syntaxe vient de sa structuration qui rend la lisibilité difficile, et donc la maintenabilité moins facile.

Pour avoir un fonctionnement plus souple, plus simple à mettre en place et plus lisible, la notion de promise a été créée.

2.1 Promise

Une promise, ou promesse en français, représente l'attente d'un résultat d'une action asynchrone. Cette notion permet de se passer des callbacks, les fonctions asynchrones n'en prenant plus en paramètre mais renvoyant une promise.

```
var todoId = 5;
var promise = recupererTodo(todoId);
```

La fonction `recupererTodo` ne prend pas de callback en paramètre mais renvoie une promise, représentant l'attente du résultat de la fonction.

Une promise peut avoir trois états. Lorsque l'action asynchrone est en cours, la promise sera dans un état d'attente de résultat. Une fois que l'action asynchrone est terminée, la promise sera soit dans un état de succès, soit dans un état d'échec. Une fois en succès ou en échec, la promise ne pourra plus changer d'état.

Un objet promise est caractérisé par une méthode `then` permettant d'agir en cas de modification de son état.

```
var promise = step1()
    .then(function(value1) {
       // Promise résolue avec succès
    }, function(error) {
       // Promise résolue avec erreur
    });
```

La méthode `then` de la promise renvoie une nouvelle promise. Cela permet de pouvoir chaîner des actions asynchrones.

```
var promise = step1()
    .then(function(value1) {
       return step2(value1);
    })
    .then(function(value2) {
       return step3(value2);
    })
```

```
    .then(function(value3) {
       return step4(value3);
    })
    .then(function (value4) {
        // Traitement lorsque l'exécution des quatre fonctions
        // asynchrones est terminée
    });
```

Le mécanisme de promise n'est pas natif dans JavaScript et n'est pas non plus lié à AngularJS. La plupart des frameworks JavaScript implémentent leur propre système de promise, comme jQuery ou WinJS.

AngularJS implémente lui aussi son propre système de promise, intégré à son architecture.

Tout le mécanisme des promises d'AngularJS est encapsulé dans le service `$q`.

2.2 Création d'une promise

Le service `$q` fournit une méthode `defer` permettant de créer un objet représentant une action asynchrone.

```
var deferred = $q.defer();
```

La propriété `promise` de l'objet renvoyé par la méthode `defer` permet d'accéder à la promise.

```
var deferred = $q.defer();

var promise = deferred.promise;
```

L'objet renvoyé par la méthode `defer` expose des méthodes permettant de modifier l'état de la promise. La méthode `resolve` résout la promise en la faisant passer dans l'état de succès. Elle prend en paramètre un objet JavaScript qui sera envoyé en tant que résultat de la promise.

```
var recupererTodo = function(id) {
    var deferred = $q.defer();

    setTimeout(function() {
           deferred.resolve({ id : id, name : "Todo " + id});
    }, 2000);

   return deferred.promise;
}
```

La fonction précédente permet de récupérer une tâche en fonction de son id. Elle crée une promise via le service `$q` et la renvoie en tant que résultat de la fonction. L'appel à la fonction `setTimeout` permet de simuler un traitement asynchrone. Au bout de deux secondes, la fonction résoudra la promise en fournissant la tâche récupérée.

La méthode `reject` permet au contraire de rejeter une promise en la faisant passer dans l'état d'erreur.

```
var recupererTodo = function(id) {
    var deferred = $q.defer();

    setTimeout(function() {
           if(id > 0) {
                  deferred.resolve({ id : id, name : "Todo " + id});
           } else {
                  deferred.reject("Id de tâche invalide");
           }
    }, 2000);

    return deferred.promise;
}
```

La fonction `recupererTodo` a été modifiée pour que la promise soit rejetée si l'id fourni en paramètre n'est pas supérieur à 0. Dans ce cas, un message d'erreur est passé en résultat de la promise.

La dernière méthode exposée par l'objet renvoyé par la méthode `defer` est `notify`. Cette méthode permet d'envoyer des informations concernant l'exécution de la fonction asynchrone en gardant la promise dans l'état d'attente de résultat.

```
var uploadItem = function(item) {
    var deferred = $q.defer();

    var progress = 0;
```

```
    var interval = $interval(function() {
           if (progress >= 100) {
               $interval.cancel(interval);
               deferred.resolve('Upload terminé');
           }

           progress += 10;
           deferred.notify(progress + '% effectué');
    }, 100);

    return deferred.promise;
}
```

La fonction précédente simule l'upload asynchrone d'un élément. La méthode `notify` de la promise permet d'envoyer un état de l'avancement de l'upload.

Le service `$q` fournit aussi une méthode `all` permettant de synchroniser l'exécution de plusieurs promises en une seule. La promise résultante sera résolue lorsque toutes les promises auront été résolues. Si au moins une des promises est rejetée, la promise résultante sera rejetée.

```
$q.all([recupererTodo(5), recupererTodo(14)]);
```

La promise renvoyée par la méthode `all` sera résolue lorsque les appels à la méthode `recupererTodo` seront terminés. Si l'un de ces appels échoue, la promise renvoyée par la méthode `all` sera rejetée.

La dernière fonction exposée par le service `$q` est la fonction `when`. Cette fonction permet d'encapsuler un objet JavaScript dans une promise. Si l'objet à encapsuler est une promise, la méthode `when` renverra simplement cette même promise.

```
var promise = $q.when(recupererTodo(5));
```

La fonction `when` renvoie la promise retournée par la fonction `recuperer-Todo`.

Si l'objet à encapsuler n'est pas une promise, la fonction `when` renverra une nouvelle promise résolue, avec comme résultat l'objet encapsulé.

```
var promise = $q.when({ id: 0, name: "Nouveau todo"});
```