Asynchronous JavaScript

1. Introduction

En JavaScript, l'asynchronisme fait référence à un style de programmation où certaines opérations peuvent être exécutées de manière non bloquante. Cela signifie que le programme ne doit pas attendre la fin d'une tâche pour passer à la suivante. Au lieu de cela, il peut planifier ces tâches en arrière-plan et continuer à exécuter d'autres parties du code.

L'asynchronisme est essentiellement utilisé pour gérer des opérations qui peuvent prendre du temps, comme l'accès à des fichiers, des appels réseau, des requêtes vers une base de données ou tout autre traitement intensif en ressources. Au lieu de bloquer l'exécution du code en attendant que ces opérations se terminent, JavaScript peut les exécuter de manière asynchrone, ce qui permet au programme de rester réactif et de continuer à répondre à d'autres interactions utilisateur ou tâches.

En JavaScript, les opérations asynchrones sont généralement gérées à l'aide de callbacks, de Promises ou plus récemment avec les fonctions asynchrones (async/await) introduites dans ECMAScript 2017. Ces mécanismes permettent aux développeurs de gérer efficacement les tâches asynchrones en contrôlant le flux d'exécution du code et en traitant les résultats une fois les opérations terminées.

1. CallBack

Un callback est une fonction passée en argument à une autre fonction pour être exécutée ultérieurement, généralement lorsque certaines opérations asynchrones sont terminées ou lorsqu'un événement spécifique se produit. Les callbacks sont largement utilisés en JavaScript pour gérer des opérations non bloquantes et pour assurer un bon contrôle du flux d'exécution du code.

Les appels non bloquants en utilisant des callbacks sont courants en JavaScript pour gérer des opérations asynchrones, telles que les requêtes réseau, l'accès aux fichiers, ou les interactions avec des bases de données. Voici un exemple simple d'appel non bloquant en utilisant un callback en Node.js avec le module **http** :

const http = require('http');

// Fonction pour effectuer une requête GET non bloquante

function faireRequeteNonBloquante(url, callback) {

  http.get(url, (response) => {

    let data = '';

    response.on('data', (chunk) => {

      data += chunk;

    });

    response.on('end', () => {

      callback(null, data); // Renvoie les données reçues au callback

    });

  }).on('error', (error) => {

    callback(error, null); // Renvoie l'erreur au callback

  });

}

// Appel de la fonction de requête non bloquante avec un callback

const url = 'http://jsonplaceholder.typicode.com/posts/1';

faireRequeteNonBloquante(url, (err, contenu) => {

  if (err) {

    console.error('Une erreur s\'est produite:', err);

  } else {

    console.log('Contenu de la requête:', contenu);

  }

});

console.log('Cette instruction s\'exécute en premier car la requête est non bloquante.');

Dans cet exemple, nous utilisons le module **http** pour effectuer une requête GET non bloquante vers l'URL **'http://jsonplaceholder.typicode.com/posts/1'**. La requête réseau se fait en arrière-plan, et nous fournissons un callback qui sera appelé une fois que la requête sera terminée.

Lorsque la réponse du serveur commence à être reçue, nous stockons progressivement les données reçues dans la variable **data**. Lorsque la réponse est entièrement reçue (**end**), nous appelons le callback avec les données reçues en second paramètre.

Si une erreur se produit pendant la requête, nous appelons le callback avec l'erreur en premier paramètre.

Comme dans l'exemple précédent, l'instruction **console.log** après l'appel de la fonction **faireRequeteNonBloquante** s'exécute immédiatement, sans attendre que la requête réseau soit terminée, car la requête est non bloquante. Une fois la requête terminée, le callback est appelé pour traiter les données ou les erreurs. C'est ainsi que les appels réseau en JavaScript sont gérés de manière non bloquante à l'aide de callbacks.

1. Promise
2. Définition

Une Promise est un objet JavaScript qui représente l'état (en attente, résolu ou rejeté) d'une opération asynchrone, et permet de gérer les résultats de cette opération de manière plus propre et efficace que les callbacks traditionnels.

Les Promises ont été introduites en ECMAScript 6 (ES6) pour simplifier la gestion des opérations asynchrones et éviter les problèmes liés aux callbacks en cascade (callback hell). Une Promise peut être dans l'un des trois états suivants :

1. En attente (pending) : L'opération asynchrone n'est pas encore terminée.
2. Résolue (fulfilled) : L'opération asynchrone a été exécutée avec succès et a renvoyé une valeur.
3. Rejetée (rejected) : L'opération asynchrone a rencontré une erreur ou un problème et a été rejetée.

Une Promise a deux méthodes importantes :

* **then()** : Cette méthode est utilisée pour spécifier ce qu'il faut faire lorsque la Promise est résolue avec succès. Elle prend deux fonctions de rappel en arguments : l'une est appelée lorsque la Promise est résolue, et l'autre est appelée lorsque la Promise est rejetée.
* **catch()** : Cette méthode est utilisée pour gérer les erreurs ou les rejets de la Promise. Elle prend une fonction de rappel qui sera exécutée si la Promise est rejetée.

Voici un exemple simple d'utilisation d'une Promise :

// Exemple de Promise pour simuler une opération asynchrone

const maPromise = new Promise((resolve, reject) => {

    setTimeout(() => {

      const randomNum = Math.random();

      if (randomNum < 0.5) {

        resolve('Succès ! Le nombre aléatoire est inférieur à 0.5.');

      } else {

        reject('Échec ! Le nombre aléatoire est supérieur ou égal à 0.5.');

      }

    }, 2000);

  });

  // Utilisation de la méthode then() pour gérer la résolution de la Promise

  maPromise.then((message) => {

    console.log(message);

  }).catch((error) => {

    console.error(error);

  });

  console.log("Cette instruction est exécutée en premier.");

Dans cet exemple, nous créons une Promise qui utilise **setTimeout** pour simuler une opération asynchrone de 2 secondes. Si le nombre aléatoire généré est inférieur à 0.5, la Promise est résolue avec un message de succès, sinon elle est rejetée avec un message d'échec.

Nous utilisons ensuite la méthode **then()** pour spécifier ce qu'il faut faire lorsque la Promise est résolue, et la méthode **catch()** pour gérer les erreurs si la Promise est rejetée. Cela permet de séparer plus proprement le traitement de la réussite et de l'échec de l'opération asynchrone.

Les Promises offrent une approche plus structurée et lisible pour gérer les opérations asynchrones en comparaison des callbacks, ce qui facilite le développement et la gestion des erreurs dans les applications JavaScript.

1. Enchaînement de Promises

Voici un exemple d'enchaînement de Promises en JavaScript pour effectuer des opérations asynchrones de manière séquentielle :

Supposons que nous ayons trois fonctions asynchrones **fonction1()**, **fonction2()**, et **fonction3()**, qui renvoient des Promises. Nous voulons les appeler de manière séquentielle, c'est-à-dire que la deuxième fonction sera appelée lorsque la première aura été résolue, et la troisième sera appelée lorsque la deuxième aura été résolue. Si l'une des Promises est rejetée, nous voulons gérer l'erreur à la fin de la chaîne

// Fonction asynchrone 1

function fonction1() {

    return new Promise((resolve, reject) => {

      setTimeout(() => {

        console.log("Fonction 1 terminée.");

        resolve(1);

      }, 1000);

    });

  }

  // Fonction asynchrone 2

  function fonction2(num) {

    return new Promise((resolve, reject) => {

      setTimeout(() => {

        console.log("Fonction 2 terminée.");

        resolve(num \* 2);

      }, 1000);

    });

  }

  // Fonction asynchrone 3

  function fonction3(num) {

    return new Promise((resolve, reject) => {

      setTimeout(() => {

        console.log("Fonction 3 terminée.");

        resolve(num \* 3);

      }, 1000);

    });

  }

  // Enchaînement des Promises

  fonction1()

    .then((resultat1) => {

      return fonction2(resultat1);

    })

    .then((resultat2) => {

      return fonction3(resultat2);

    })

    .then((resultatFinal) => {

      console.log("Résultat final :", resultatFinal);

    })

    .catch((erreur) => {

      console.error("Une erreur s'est produite :", erreur);

    });

Dans cet exemple, chaque fonction asynchrone renvoie une Promise qui est résolue après un délai de 1 seconde (simulé avec **setTimeout**). Lorsque la première fonction est résolue, nous utilisons **then()** pour chaîner l'appel à la deuxième fonction, et ainsi de suite jusqu'à la troisième fonction. Finalement, dans la dernière **then()**, nous obtenons le résultat final de la chaîne d'opérations.

Si l'une des Promises est rejetée à n'importe quel stade de la chaîne, l'erreur est capturée et gérée par la méthode **catch()** à la fin de la chaîne. Cela garantit que toute erreur survenue dans l'une des fonctions asynchrones est correctement prise en charge sans interrompre le flux d'exécution de la chaîne d'opérations.

1. Fetch

Voici un exemple d'utilisation de **fetch** en JavaScript pour effectuer une requête HTTP pour récupérer des données à partir d'une API. **fetch** est une fonction JavaScript native introduite dans les navigateurs modernes et dans l'environnement Node.js via des modules complémentaires, tels que **node-fetch**, pour effectuer des requêtes réseau asynchrones.

Dans cet exemple, nous allons utiliser **fetch** pour récupérer des données JSON depuis l'API "JSONPlaceholder", qui fournit des données de test pour les requêtes HTTP.

Assurez-vous d'exécuter cet exemple dans un environnement qui prend en charge **fetch**. Dans le navigateur, cela fonctionnera par défaut. Dans Node.js, vous pouvez installer le module **node-fetch** pour bénéficier du même comportement.

Voici le code :

// Dans un environnement Node.js, décommentez la ligne ci-dessous pour utiliser node-fetch

// const fetch = require('node-fetch');

// URL de l'API JSONPlaceholder pour récupérer des données factices

const apiUrl = 'https://jsonplaceholder.typicode.com/posts/1';

// Utilisation de fetch pour effectuer une requête GET

fetch(apiUrl)

  .then(response => {

    // Vérifier si la réponse a été réussie (statut 200-299)

    if (!response.ok) {

      throw new Error('La requête a échoué avec le statut ' + response.status);

    }

    // Récupérer les données JSON renvoyées par l'API

    return response.json();

  })

  .then(data => {

    // Traitement des données JSON

    console.log('Données récupérées :', data);

  })

  .catch(error => {

    console.error('Une erreur s\'est produite :', error);

  });

Dans cet exemple, nous utilisons **fetch** pour effectuer une requête GET vers l'API JSONPlaceholder. La méthode **fetch** renvoie une Promise qui représente la réponse HTTP. Nous utilisons les méthodes **then()** pour traiter la réponse et récupérer les données JSON.

Le premier **then()** vérifie si la réponse a été réussie (statut HTTP 200-299) en vérifiant **response.ok**. Si la réponse est valide, nous appelons **response.json()** pour extraire les données JSON de la réponse. Sinon, nous rejetons la Promise avec une erreur personnalisée.

Dans le deuxième **then()**, nous pouvons traiter les données JSON renvoyées par l'API. Dans cet exemple, nous nous contentons de les afficher dans la console.

Si une erreur se produit pendant le processus de requête ou de traitement des données, la méthode **catch()** gère l'erreur et l'affiche dans la console.

Cet exemple illustre comment utiliser **fetch** pour effectuer des requêtes HTTP et récupérer des données depuis une API de manière asynchrone en JavaScript.

1. Async Await

**async/await** est une fonctionnalité introduite en ECMAScript 2017 (ES8) pour simplifier la gestion des opérations asynchrones en JavaScript. Cela permet de rendre le code asynchrone plus lisible, plus proche de la syntaxe séquentielle, sans utiliser explicitement les Promises ou les callbacks.

Lorsque vous utilisez **async/await**, vous pouvez marquer une fonction avec le mot-clé **async**, ce qui signifie que cette fonction retournera toujours une Promise. À l'intérieur de cette fonction, vous pouvez utiliser le mot-clé **await** pour attendre la résolution d'une Promise. Cela permet d'effectuer des opérations asynchrones de manière séquentielle, comme si elles étaient synchrones.

Voici un exemple d'utilisation d'**async/await** pour effectuer des opérations asynchrones avec **fetch** :

// Dans un environnement Node.js, décommentez la ligne ci-dessous pour utiliser node-fetch

// const fetch = require('node-fetch');

// URL de l'API JSONPlaceholder pour récupérer des données factices

const apiUrl = 'https://jsonplaceholder.typicode.com/posts/1';

// Fonction asynchrone pour récupérer les données depuis l'API

async function recupererDonnees() {

  const response = await fetch(apiUrl);

  // Vérifier si la réponse a été réussie (statut 200-299)

  if (!response.ok) {

    throw new Error('La requête a échoué avec le statut ' + response.status);

  }

  // Récupérer les données JSON renvoyées par l'API

  return response.json();

}

// Appel de la fonction asynchrone avec async/await et .catch()

recupererDonnees()

  .then(donnees => {

    console.log('Données récupérées :', donnees);

  })

  .catch(error => {

    console.error('Une erreur s\'est produite :', error);

  });

Dans cet exemple, la fonction **recupererDonnees()** est marquée comme **async**, ce qui signifie qu'elle renverra toujours une Promise.

Nous utilisons **await** pour attendre la résolution de la Promise retournée par **fetch()**. Si une erreur se produit, la Promise sera rejetée et la fonction **.catch()** sera appelée dans le bloc d'appel de la fonction asynchrone.

Cela permet de gérer les erreurs de manière similaire à l'utilisation de **.catch()** avec des Promises traditionnelles, tout en bénéficiant de la simplicité et de la lisibilité de la syntaxe **async/await**.

Notez que si vous n'utilisez pas un bloc **try/catch** ou la méthode **.catch()**, les erreurs survenant dans la fonction asynchrone seront silencieuses et peuvent ne pas être correctement gérées. Il est donc généralement préférable de gérer les erreurs pour éviter des comportements inattendus dans votre application.

1. Try catch

**try/catch** est une construction en JavaScript utilisée pour gérer les erreurs dans le code de manière explicite. Lorsqu'une opération peut générer une exception (une erreur), vous pouvez l'entourer avec un bloc **try** pour essayer (try) d'exécuter cette opération, et s'il y a une exception, vous pouvez la capturer et la gérer dans le bloc **catch**.

La syntaxe générale est la suivante :

try {

    // Code susceptible de générer une exception (erreur)

    // ...

  } catch (erreur) {

    // Gestion de l'erreur

    // ...

  }

Lorsque le code à l'intérieur du bloc **try** génère une exception, le flux d'exécution est interrompu et le contrôle est transféré au bloc **catch**. L'erreur est capturée dans le paramètre **erreur**, que vous pouvez utiliser pour afficher des informations sur l'erreur ou effectuer des actions de récupération.

Voici un exemple concret :

function diviser(a, b) {

    try {

      if (b === 0) {

        throw new Error("Division par zéro !");

      }

      return a / b;

    } catch (erreur) {

      console.error("Une erreur s'est produite :", erreur.message);

      return NaN;

    }

  }

  const resultat1 = diviser(10, 2);

  console.log("Résultat 1 :", resultat1); // Affiche : Résultat 1 : 5

  const resultat2 = diviser(10, 0);

  console.log("Résultat 2 :", resultat2); // Affiche : Une erreur s'est produite : Division par zéro ! / Résultat 2 : NaN

Dans cet exemple, la fonction **diviser()** effectue une division entre **a** et **b**. Si **b** est égal à zéro, nous lançons une exception avec **throw** pour indiquer une division par zéro. Dans le bloc **catch**, nous affichons un message d'erreur personnalisé et retournons **NaN** pour signaler que le résultat est indéterminé.

Le bloc **try/catch** permet de gérer explicitement les erreurs et d'éviter que des exceptions non contrôlées interrompent le programme. Il est recommandé d'utiliser **try/catch** lorsqu'une opération est susceptible de générer une exception et que vous souhaitez gérer cette situation de manière appropriée.

1. Node
2. **Flux de contrôle**

Une image contenant texte

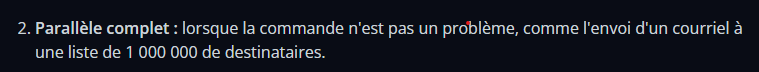
Description générée automatiquement

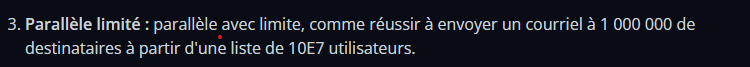
Une image contenant texte

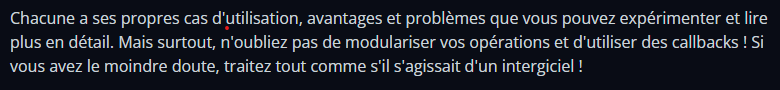
Description générée automatiquement

**Une image contenant texte

Description générée automatiquement**

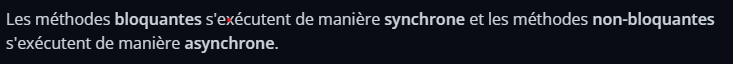
****





1. **Vue d’ensemble du blocage et du non-blocage**

Cette présentation couvre la différence entre les appels bloquants et non-bloquants dans Node.js. Cet aperçu fera référence à la boucle d'événement et à libuv mais aucune connaissance préalable de ces sujets n'est requise**.**

****

**Une image contenant texte

Description générée automatiquement**

Une image contenant texte

Description générée automatiquement

En Node.js, **process.nextTick()** est une méthode qui permet de planifier l'exécution d'une fonction pour le prochain cycle d'événement du processus Node.js. Contrairement à la méthode **setImmediate()**, qui planifie l'exécution d'une fonction pour le prochain cycle d'événement du bouclage d'événements (event loop), **process.nextTick()** planifie l'exécution d'une fonction pour le cycle d'événement suivant du processus.

L'utilisation de **process.nextTick()** permet de reporter l'exécution d'une fonction jusqu'à ce que toutes les opérations synchrones en cours soient terminées, ce qui peut améliorer les performances et éviter les blocages de boucle d'événements. Cela peut également aider à éviter les problèmes de synchronisation liés à l'utilisation de callbacks et de promesses.

En résumé, **process.nextTick()** est une méthode utile pour la programmation asynchrone en Node.js, car elle permet de planifier l'exécution d'une fonction pour le prochain cycle d'événement du processus, ce qui peut améliorer les performances et éviter les blocages.

Une image contenant texte

Description générée automatiquement

Une image contenant texte, Logiciel multimédia, logiciel, Logiciel de graphisme

Description générée automatiquement

1. **Les minuteur Js**

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte, capture d’écran, écran

Description générée automatiquement

Une image contenant texte, capture d’écran, écran, noir

Description générée automatiquement

Une image contenant texte, capture d’écran, écran, fermer

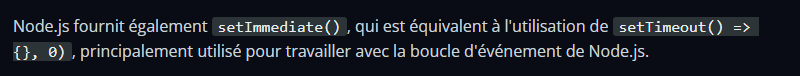
Description générée automatiquement

Une image contenant texte, capture d’écran, écran

Description générée automatiquement

Une image contenant texte, capture d’écran, écran, noir

Description générée automatiquement



1. **setImmediate**

Une image contenant texte, capture d’écran, écran

Description générée automatiquement

Une image contenant texte, capture d’écran, écran, noir

Description générée automatiquement

1. L’émetteur d’évènements de nodeJs

Une image contenant texte, capture d’écran, noir, écran

Description générée automatiquement

Une image contenant texte, capture d’écran, écran

Description générée automatiquement