28 SEPTEMBRE 2020 / #JAVASCRIPT

Tutoriel JavaScript Async Await - Comment attendre la fin d'une fonction dans JS

Fredrik Strand Oseberg

Quand une fonction asynchrone se termine-t-elle? Et pourquoi est-ce une question si difficile à répondre?

Eh bien, il s'avère que la compréhension des fonctions asynchrones nécessite beaucoup de connaissances sur le fonctionnement fondamental de JavaScript.

Allons explorer ce concept et apprenons beaucoup de choses sur JavaScript dans le processus.

Es-tu prêt? Allons-y.

Qu'est-ce que le code asynchrone?

De par sa conception, JavaScript est un langage de programmation synchrone. Cela signifie que lorsque le code est exécuté, JavaScript commence en haut du fichier et parcourt le code ligne par ligne, jusqu'à ce que cela soit fait.

Le résultat de cette décision de conception est qu'une seule chose peut se produire à la fois.

Vous pouvez penser à cela comme si vous jongliez avec six petites balles. Pendant que vous jonglez, vos mains sont occupées et ne peuvent rien gérer d'autre.

C'est la même chose avec JavaScript: une fois que le code est en cours d'exécution, il a les mains pleines avec ce code. Nous appelons cela ce type de *blocage de* code synchrone . Parce que cela bloque efficacement l'exécution d'autres codes.

Revenons à l'exemple de la jonglerie. Que se passerait-il si vous vouliez ajouter une autre balle? Au lieu de six balles, vous vouliez jongler avec sept balles. Cela pourrait être un problème.

Vous ne voulez pas arrêter de jongler, car c'est tellement amusant. Mais vous ne pouvez pas non plus aller chercher une autre balle, car cela signifierait que vous deviez vous arrêter.

La solution? Déléguez le travail à un ami ou à un membre de la famille. Ils ne jonglent pas, alors ils peuvent aller chercher le ballon pour vous, puis le lancer dans votre jonglerie à un moment où votre main est libre et que vous êtes prêt à ajouter une autre balle au milieu de la jonglerie.

C'est ce qu'est le code asynchrone. JavaScript délègue le travail à autre chose, puis s'occupe de sa propre entreprise. Ensuite, quand il sera prêt, il recevra les résultats du travail.

Qui fait l'autre travail?

Très bien, nous savons donc que JavaScript est synchrone et paresseux. Il ne veut pas faire tout le travail lui-même, alors il le ferme à autre chose.

Mais qui est cette entité mystérieuse qui fonctionne pour JavaScript? Et comment est-il embauché pour travailler pour

JavaScript? Et comment est-il embauché pour travailler pour

JavaScript?

Eh bien, jetons un coup d'œil à un exemple de code asynchrone.

```
const logName = () => {
    console.log("Han")
}

setTimeout(logName, 0)

console.log("Hi there")
```

L'exécution de ce code entraîne la sortie suivante dans la console:

```
// in console
Hi there
Han
```

Bien. Que se passe-t-il?

Il s'avère que la façon dont nous travaillons dans JavaScript consiste à

utiliser des fonctions et des API spécifiques à l'environnement. Et c'est une source de grande confusion en JavaScript.

JavaScript s'exécute toujours dans un environnement.

Souvent, cet environnement est le navigateur. Mais cela peut aussi être sur le serveur avec NodeJS. Mais quelle est la différence?

La différence - et c'est important - est que le navigateur et le serveur (NodeJS), du point de vue des fonctionnalités, ne sont pas équivalents. Ils sont souvent similaires, mais ils ne sont pas identiques.

Illustrons cela avec un exemple. Disons que JavaScript est le protagoniste d'un livre fantastique épique. Juste un enfant de ferme ordinaire.

Disons maintenant que ce gamin de la ferme a trouvé deux armures spéciales qui leur ont donné des pouvoirs au-delà des leurs.

Lorsqu'ils utilisaient l'armure du navigateur, ils avaient accès à un certain ensemble de capacités.

Lorsqu'ils ont utilisé l'armure du serveur, ils ont eu accès à un autre ensemble de capacités.

Ces combinaisons se chevauchent, car les créateurs de ces combinaisons avaient les mêmes besoins à certains endroits, mais pas à d'autres.

Voilà ce qu'est un environnement. Un endroit où le code est exécuté, où il existe des outils basés sur le langage JavaScript existant. Ils ne font pas partie du langage, mais la ligne est souvent floue car nous utilisons ces outils tous les jours lorsque nous écrivons du code.

setTimeout , fetch et DOM sont tous des exemples d'API Web. (Vous pouvez voir la liste complète des API Web ici.) Ce sont des outils intégrés au navigateur et mis à notre disposition lorsque notre code est exécuté.

Et comme nous exécutons toujours JavaScript dans un environnement, il semble que ceux-ci font partie du langage. Mais ils ne le sont pas.

Donc, si vous vous êtes déjà demandé pourquoi vous pouvez utiliser fetch en JavaScript lorsque vous l'exécutez dans le navigateur (mais que vous devez installer un package lorsque vous l'exécutez dans NodeJS), voici pourquoi. Quelqu'un a pensé que Fetch était une bonne idée et l'a construit comme un outil pour l'environnement NodeJS.

Déroutant? Oui!

Déroutant? Oui.

Mais maintenant, nous pouvons enfin comprendre ce qui prend le travail de JavaScript et comment il est embauché.

Il s'avère que c'est l'environnement qui prend en charge le travail, et la façon de faire en sorte que l'environnement fasse ce travail, est d'utiliser des fonctionnalités qui appartiennent à l'environnement. Par exemple, *fetch* ou *setTimeout* dans l'environnement du navigateur.

Qu'arrive-t-il au travail?

Génial. Donc, l'environnement prend le travail. Alors quoi?

À un moment donné, vous devez récupérer les résultats. Mais réfléchissons à la façon dont cela fonctionnerait.

Revenons à l'exemple de la jonglerie du début. Imaginez que vous ayez demandé un nouveau ballon et qu'un ami ait juste commencé à vous lancer le ballon alors que vous n'étiez pas prêt.

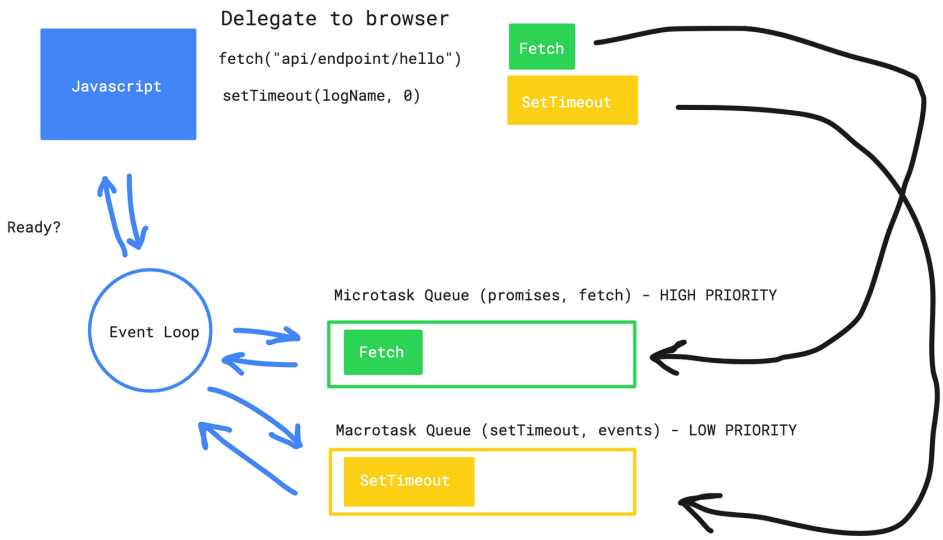
Ce serait un désastre. Peut-être que vous pourriez avoir de la chance et l'attraper et l'intégrer efficacement dans votre routine. Mais il y a de

grandes chances que cela vous fasse perdre toutes vos balles et que

votre routine échoue. Ne serait-il pas préférable de donner des instructions strictes sur le moment de recevoir le ballon?

En fait, il existe des règles strictes concernant le moment où JavaScript peut recevoir du travail délégué.

Ces règles sont régies par la boucle d'événements et impliquent la file d'attente microtâche et macrotask. Oui je sais. C'est beaucoup. Mais supportez-moi.



Bien. Ainsi, lorsque nous déléguons du code asynchrone au navigateur, le navigateur prend et exécute le code et prend en charge

navigateur, le navigateur prend et exécute le code et prend en charge cette charge de travail. Mais il peut y avoir plusieurs tâches qui sont confiées au navigateur, nous devons donc nous assurer que nous pouvons hiérarchiser ces tâches.

C'est là que la file d'attente des microtaches et la file des macrotask entrent en jeu. Le navigateur prendra le travail, le fera, puis placera le résultat dans l'une des deux files d'attente en fonction du type de travail qu'il reçoit.

Les promesses, par exemple, sont placées dans la file d'attente de microtâche et ont une priorité plus élevée.

Les événements et setTimeout sont des exemples de travail mis dans la file d'attente de macrotask et ont une priorité inférieure.

Maintenant, une fois le travail terminé et placé dans l'une des deux files d'attente, la boucle d'événements s'exécutera d'avant en arrière et vérifiera si JavaScript est prêt à recevoir les résultats.

Ce n'est que lorsque JavaScript a terminé d'exécuter tout son code synchrone, et qu'il est bon et prêt, que la boucle d'événements commencera à choisir dans les files d'attente et à remettre les fonctions à JavaScript pour qu'elles s'exécutent.

Jetons donc un œil à un exemple:

```
setTimeout(() => console.log("hello"), 0)

fetch("https://someapi/data").then(response => re
                              .then(data => consol

console.log("What soup?")
```

Quelle sera la commande ici?

1. Tout d'abord, setTimeout est délégué au navigateur, qui fait le travail et place la fonction résultante dans la file d'attente des macrotask.
2. Deuxièmement, la récupération est déléguée au navigateur, qui prend le travail. Il récupère les données du point de terminaison et place les fonctions résultantes dans la file d'attente de microtâche.
3. Javascript déconnecte "Quelle soupe"?
4. La boucle d'événements vérifie si JavaScript est prêt à recevoir les résultats du travail mis en file d'attente.
5. Lorsque le fichier console.log est terminé, JavaScript est prêt. La boucle d'événements sélectionne les fonctions en file d'attente dans la file d'attente de microtâche, qui a une priorité plus élevée, et les renvoie à JavaScript pour qu'elles s'exécutent.
6. Une fois la file d'attente de microtâche vide, le rappel setTimeout est retiré de la file d'attente de macrotask et renvoyé à JavaScript pour qu'il s'exécute.

```
In console:
// What soup?
// the data from the api
// hello
```

Promesses

Vous devriez maintenant avoir une bonne connaissance de la manière dont le code asynchrone est géré par JavaScript et l'environnement du navigateur. Alors parlons de promesses.

Une promesse est une construction JavaScript qui représente une future valeur inconnue. Conceptuellement, une promesse n'est que JavaScript promettant de renvoyer *une valeur* . Il peut s'agir du résultat d'un appel d'API ou d'un objet d'erreur d'une requête réseau ayant échoué. Vous êtes assuré d'obtenir quelque chose.

```
const promise = new Promise((resolve, reject) =>
        // Make a network request
   if (response.status === 200) {
      resolve(response.body)
```

```
    } else {
        const error = { ... }
        reject(error)
    }
})

promise.then(res => {

        console.log(res)
}).catch(err => {
        console.log(err)
})
```

Une promesse peut avoir les états suivants:

- rempli - action réussie
- rejeté - l'action a échoué
- en attente - aucune action n'a été effectuée
- réglé - a été exécuté ou rejeté

Une promesse reçoit une fonction de résolution et de rejet qui peut être appelée pour déclencher l'un de ces états.

L'un des grands arguments de vente des promesses est que nous pouvons enchaîner les fonctions que nous souhaitons voir se produire en cas de succès (résolution) ou d'échec (rejet):

- Pour enregistrer une fonction à exécuter en cas de succès, nous utilisons .then
- Pour enregistrer une fonction à exécuter en cas d'échec, nous utilisons .catch

```
// Fetch returns a promise

fetch("https://swapi.dev/api/people/1")
      .then((res) => console.log("This function is run when the request s
      .catch(err => console.log("This function is run when the request fa

// Chaining multiple functions
 fetch("https://swapi.dev/api/people/1")
      .then((res) => doSomethingWithResult(res))
      .then((finalResult) => console.log(finalResult))
      .catch((err => doSomethingWithErr(err))
```

Parfait. Examinons maintenant de plus près à quoi cela ressemble sous le capot, en utilisant fetch comme exemple:

```
const fetch = (url, options) => {
   // simplified
   return new Promise((resolve, reject) => {
```

```
  const xhr = new XMLHttpRequest()
  // ... make request
  xhr.onload = () => {
    const options = {
      status: xhr.status,
      statusText: xhr.statusText
      ...
    }

    resolve(new Response(xhr.response, options))
  }

  xhr.onerror = () => {
    reject(new TypeError("Request failed"))
  }
}

fetch("https://swapi.dev/api/people/1")
  // Register handleResponse to run when promise resolves
  .then(handleResponse)
  .catch(handleError)

// conceptually, the promise looks like this now:
// { status: "pending", onsuccess: [handleResponse], onfailure: [handleError]

const handleResponse = (response) => {
  // handleResponse will automatically receive the response, ¨
  // because the promise resolves with a value and automatically injects into
  console.log(response)
```

```
  }

    const handleError = (response) => {
    // handleError will automatically receive the error, ¨
    // because the promise resolves with a value and automatically injects into
      console.log(response)
  }

// the promise will either resolve or reject causing it to run all of the regis

// injecting the value. Let's inspect the happy path:

// 1. XHR event listener fires
// 2. If the request was successfull, the onload event listener triggers
// 3. The onload fires the resolve(VALUE) function with given value
// 4. Resolve triggers and schedules the functions registered with .then
```

Nous pouvons donc utiliser les promesses pour faire un travail asynchrone et être sûrs de pouvoir gérer tout résultat de ces promesses. Telle est la proposition de valeur. Si vous voulez en savoir plus sur les promesses, vous pouvez en savoir plus ici et ici .

Lorsque nous utilisons des promesses, nous enchaînons nos fonctions sur la promesse de gérer les différents scénarios.

Cela fonctionne, mais nous devons toujours gérer notre logique à l'intérieur des callbacks (fonctions imbriquées) une fois que nous récupérons nos résultats. Et si nous pouvions utiliser des promesses mais écrire du code synchrone? Il s'avère que nous pouvons.

Asynchrone / Attendre

Async / Await est une manière d'écrire des promesses qui nous permet d'écrire du *code asynchrone de manière synchrone.* Regardons.

```
const getData = async () => {
    const response = await fetch("https://jsonpla
    const data = await response.json()

    console.log(data)
}

getData()
```

Rien n'a changé sous le capot ici. Nous utilisons toujours les

promesses pour récupérer les données, mais elles semblent désormais synchrones et nous n'avons plus de blocs .then et .catch.

Async / Await n'est en fait que du sucre syntaxique fournissant un moyen de créer du code plus facile à raisonner, sans changer la dynamique sous-jacente.

Jetons un coup d'œil à son fonctionnement.

Async / Await nous permet d'utiliser
des générateurs pour *suspendre* l'exécution d'une fonction. Lorsque nous utilisons async / await, nous ne bloquons pas car la fonction cède le contrôle au programme principal.

Ensuite, lorsque la promesse se résout, nous utilisons le générateur pour céder le contrôle à la fonction asynchrone avec la valeur de la promesse résolue.

Vous pouvez en savoir plus ici pour un excellent aperçu des générateurs et du code asynchrone.

En effet, nous pouvons maintenant écrire du code asynchrone qui ressemble à du code synchrone. Ce qui signifie qu'il est plus facile de

raisonner et que nous pouvons utiliser des outils synchrones pour la gestion des erreurs tels que try / catch:

```
const getData = async () => {
      try {
            const response = await fetch("https://jsonplaceholder.typic
            const data = await response.json()
            console.log(data)
      } catch (err) {
          console.log(err)
      }

}

getData()
```

Bien. Alors, comment l'utilisons-nous? Pour utiliser async / await, nous devons ajouter la fonction async. Cela n'en fait pas une fonction asynchrone, cela nous permet simplement d'utiliser wait à l'intérieur de celle-ci.

Le fait de ne pas fournir le mot-clé async entraînera une erreur de syntaxe lors de la tentative d'utilisation de wait dans une fonction normale.

```
const getData = async () => {
      console.log("We can use await in this function")
}
```

Pour cette raison, nous ne pouvons pas utiliser async / await sur le code de niveau supérieur. Mais async et wait ne sont encore que du sucre syntaxique sur les promesses. Nous pouvons donc gérer les cas de haut niveau avec le chaînage des promesses:

```
async function getData() {
  let response = await fetch('http://apiurl.com')
}

// getData is a promise
getData().then(res => console.log(res)).catch(err
```

Cela expose un autre fait intéressant sur async / await. Lors de la définition d'une fonction asynchrone, *elle renverra toujours une promesse.*

Utiliser async / await peut sembler magique au premier abord. Mais comme toute magie, c'est juste une technologie suffisamment avancée qui a évolué au fil des ans. J'espère que vous avez maintenant une solide compréhension des fondamentaux et que vous pouvez utiliser async / await en toute confiance.

Conclusion

Si vous avez réussi ici, félicitations. Vous venez d'ajouter à votre boîte à outils un élément clé de connaissances sur JavaScript et son fonctionnement avec ses environnements.

C'est certainement un sujet déroutant, et les lignes ne sont pas toujours claires. Mais maintenant, j'espère que vous avez une compréhension du fonctionnement de JavaScript avec du code asynchrone dans le navigateur, et une meilleure compréhension des promesses et de l'async / wait.

Si vous avez apprécié cet article, vous pourriez également profiter de ma chaîne youtube . J'ai actuellement une série sur les fondamentaux du Web où je passe par HTTP , en créant des serveurs Web à partir de zéro et plus encore.

Il y a aussi une série sur la création d'une application entière avec React , si c'est votre confiture. Et je prévois d'ajouter beaucoup plus de contenu ici à l'avenir pour approfondir les sujets JavaScript.

Et si vous voulez dire bonjour ou discuter du développement Web, vous pouvez toujours me contacter sur Twitter à @foseberg. Merci d'avoir lu!