5 minutes

Event loop

Cette leçon est très importante car elle va vous permettre de comprendre tout ce chapitre en apprenant comment il est possible de faire ces traitements asynchrones en `JavaScript` en restant non bloquants.

Autrement dit, nous allons voir comment le moteur `JavaScript` fait pour gérer des traitements asynchrones alors que nous avons vu qu'il n'y avait qu'un seul fil d'exécution (`thread`).

La boucle d'événements (`Event loop`)

La boucle d'événements (`event loop`) est une boucle infinie qui permet de gérer tous les événements.

Nous avons besoin de voir un certain nombre d'éléments avant de tout comprendre.

Les `Web APIs` et la file des tâches

Les `Web APIs` asynchrones utilisent des fils d'exécution (`thread`) distinctes du `thread` principal du moteur `JavaScript` qui exécute la pile d'exécution (la `stack`).

Ils utilisent des `threads` du navigateur pour réaliser leurs tâches asynchrones (envoyer une requête etc).

Les `Web APIs` peuvent être les `timers` (`setTimeout()`), des requêtes `HTTP` (`fetch` ou `XMLHttpRequest` que nous allons étudier), des événements du `DOM` (`addEventListener()`), et plein d'autres choses qui peuvent être asynchrones.

Lorsque la tâche asynchrone est terminée, le gestionnaire de la tâche (par exemple le gestionnaire d'événement passé à `addEventListener()`) va être placé sur la file des tâches.

La file des tâches

La file des tâches contient toutes les fonctions passées comme gestionnaire à une `Web API` asynchrone et qui a terminé son traitement.

Il s'agit des fonctions de rappels par exemple les fonctions 1 et 2 :

```
setTimeout(fonction1, 2000);
document.addEventListener("click", fonction2);
```

Lorsque le `timer` a terminé la fonction1 est ajoutée à la pile des tâches. De même, si un clic se produit, la fonction2 est ajoutée à la pile des tâches.

Lorsque la pile d'exécution est vide, le moteur va vérifier qu'il y a des tâches sur la pile des tâches . Si il y en a il va en prendre une seule et l'exécuter puis recommencer un cycle (voir plus bas).

La file des tâche fonctionne selon le principe `FIFO`, et donc la première tâche qui a été ajoutée est la première a être exécutée lorsque la pile est vide.

La file des micro-tâches

Les promesses utilisent une file différente qui est aussi une file `FIFO` (`first-in-first-out`) : la file des micro-tâches.

Ce qui se passe est que les gestionnaires de promesses sont mis sur cette file lorsque la promesse est acquittée.

Ainsi, lorsque nous appelons `then()`, cette méthode est ajoutée à la file des micro tâches lorsque la promesse sur laquelle elle est appelée est acquittée.

D'autres exemples d'objets et de fonctions utilisant cette file spéciale sont : `setImmediate()`, `MutationObserver` et `process.nextTick` que nous étudierons plus tard.

Les priorités des différentes tâches

Pour bien comprendre l'ordre d'exécution des tâches il faut comprendre l'ordre de priorité suivant :

Pile d'exécution > file des micro-tâches > file des tâches.

Une fois la pile d'exécution vide, le moteur va regarder la file des micro-tâches et toutes les exécuter.

Ensuite, il va regarder la file des tâches et en retirer une seule et l'exécuter.

Ensuite, il va vider la pile d'exécution et vider la file des micro-tâches et reprendre une seule tâche de la file.

Lorsque tout est exécuté il va attendre que d'autres tâches soient ajoutées grâce à la boucle infinie : l'`event loop`.

Le rendu

Il faut aussi afficher visuellement les résultats des traitements c'est le rôle du rendu. Le navigateur doit rafraîchir l'écran environ 60 fois par seconde pour que l'expérience utilisateur soit fluide.

La file des rendus (modifications à effectuer à l'écran) est exécutée juste après la file des micro-tâches et est aussi exécutée jusqu'à qu'il n'y ait plus rien dans la file.

Résumé de l'ordre de l'`event loop`

On part du principe que la pile d'exécution est vide, tout le `JavaScript` synchrone ayant été exécuté.

1 - La tâche la plus ancienne, ajoutée en première sur la file des tâches, est retirée de la file et exécutée.

2 - La file des micro-tâches est exécutée dans l'ordre `FIFO`.

3 - La file des rendus est exécutée.

4 - Si la file des tâches est vide attendre, sinon retour à l'étape 1.

Exemples

Vous pouvez essayez de réfléchir à cet exemple puis voir la solution et faire des tests avec l'exécutable :

```
setTimeout(() => afficherLeResultat("Le timeout"), 0);
Promise.resolve().then(() => afficherLeResultat("La promesse 1"));
requestAnimationFrame(() => afficherLeResultat("L'animation"));
Promise.resolve().then(() => afficherLeResultat("La promesse 2"));
afficherLeResultat("Le code synchrone");
```

Solution :
https://codesandbox.io/embed/js-c14-l7-1-5ff40

Vous pouvez essayez de réfléchir à cet exemple puis voir la solution et faire des tests avec l'exécutable :

```
const button = document.querySelector("button");
button.addEventListener("click", () => {
  Promise.resolve().then(() => afficherLeResultat("La promesse 1"));
  afficherLeResultat("Le clic 1");
});
button.addEventListener("click", () => {
  Promise.resolve().then(() => afficherLeResultat("La promesse 2"));
  afficherLeResultat("Le clic 2");
});
```

La solution :
https://codesandbox.io/embed/js-c14-l7-2-6u6nw

Rappelez les vous que les gestionnaires d'événements passés à `addEventListener()` sont des tâches. L'ordre est donc le suivant lors d'un clic :

1 - Deux gestionnaires d'événement sont sur la file des tâches.

2 - Une première tâche est prise sur la file des tâches et exécutée. La fonction asynchrone est exécutée puis la file des micro-tâches qui contient le gestionnaire de la promesse.

3 - La deuxième tâche est prise sur la file et exécutée. La fonction asynchrone est exécutée puis la file des micro-tâches qui contient le gestionnaire de la promesse.