Les événements

Les événements en JavaScript

La boucle d’événements

JavaScript gère la concurrence grâce à une boucle d'événements.
Pour comprendre le fonctionnement de cette boucle d’événements, il est important de comprendre les notions suivantes.

La pile d’appels (stack)
Les appels de fonction forment une pile de cadre (frames).
A chaque appel de fonction, on empile les arguments de la fonction ainsi que ses variables locales.

Le tas (heap)
Les objets sont alloués dans une zone mémoire qu’on appelle le tas.

La file (queue)
L’environnement d’exécution JavaScript (runtime) contient une queue de messages à traiter.
Lorsque la pile est vide, on retire un message de la queue et on le traite. Le traitement consiste à appeler la fonction associée au message. Le traitement d'un message est fini lorsque la pile d'appels redevient vide.

La boucle d’événements

Le pseudo-code de la boucle d’événement ressemble à

```
while (queue.attendreMessage()){
  queue.traiterProchainMessage();
}
```

Un modèle non bloquant

Le modèle de la boucle d'événement ne bloque jamais.
La gestion des entrées-sorties (I/O) est généralement traitée par des événements et des callbacks. Ainsi quand l’application attend un résultat, il est possible de traiter d’autres éléments comme les saisies utilisateur.

Schéma de la boucle d'événements

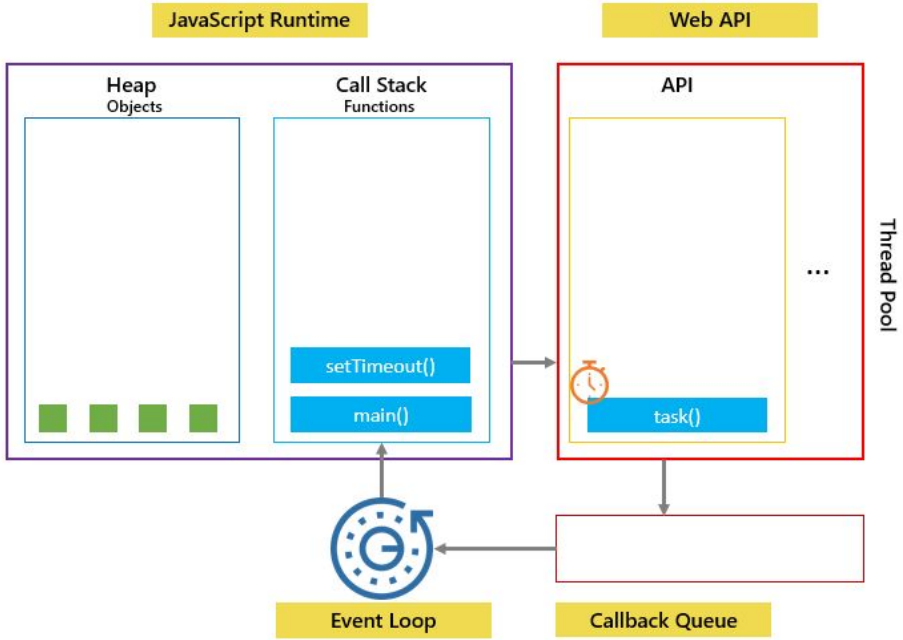

Schéma de la boucle d’événements

Simulateur de la boucle d’événements en ligne.

http://latentflip.com/loupe

Démonstration du fonctionnement de la boucle d’événements

Gestion des événements

Supposons que nous ayons un bouton dont l'id='envoi' et une fonction 'handleClick' qui traite l'événement du clic.

```
<button id="envoi">Envoi</button>
```

Il y a 3 manières différentes de gérer l'événement de clic.

1. Utilisation de l'attribut 'onclick' du bouton
   ```
   <button id="envoi" onclick="handleClick()">Envoi</button>
   ```
2. Utilisation de la propriété 'onclick' de l'élément
   ```
   document.getElementById("envoi").onclick = handleClick;
   ```
3. Ajout d'un event listener sur l'élément
   ```
   document.getElementById("envoi").addEventListener('click', handleClick);
   ```

Avec la fonction 'addEventListener', nous pouvons attacher plusieurs fonctions différentes au même événement.

Gestion des événements

L'attribut onclick peut prendre 2 arguments

```
<button id="envoi" onclick="handleClick(event, this)">
```

event: correspond à l'événement à l'origine de l'appel de la fonction.

event.target est l'objet à l'origine de cet événement.

this: correspond au noeud HTML courant. Celui qui déclare le gestionnaire de l'événement (event handler).

Les gestionnaires d'événements déclarés à l'aide de la propriété 'onclick' ainsi que ceux déclarés à l'aide de 'AddEventListener' peuvent également prendre un paramètre 'event'.

```
document.getElementById("envoi").onclick = (event) => console.log(event.target.id);
```

```
document.getElementById("envoi").addEventListener('click', (event) => console.log(event.target.id));
```

Capture et bubbling de l’événement

La capture (capturing) et le bubbling sont 2 modes de propagation d’événements dans le DOM.

Lorsqu’un événement se produit dans un élément à l'intérieur d'un autre élément, et que les deux éléments ont déclaré un gestionnaire d’événement pour cet événement, le mode de propagation détermine dans quel ordre les éléments reçoivent l’événement.

Dans le cas du Bubbling, l'événement est d'abord capturé et traité par l'élément le plus à l'intérieur, puis propagé aux éléments extérieurs.

Dans le cas de la capture (capturing), l'événement est d'abord capturé par l'élément le plus à l'extérieur et propagé aux éléments intérieurs.

Capture et bubbling de l’événement

Pour le DOM suivant

L’événement bubbling est traité dans cet ordre p > div > form

L’événement de capture (capturing) est traité dans cet ordre form > div > p

Gestion des événements

Comme évoqué précédemment, le modèle de la boucle d'événement ne bloque jamais.

Un événement n’est pas traité par la pile d’appels (call stack) courante. Lorsque l’événement se déclenche, il est mis dans la queue des callbacks. C’est la boucle d’événement qui est chargée de récupérer les fonctions callbacks une que notre call stack est vide.

En JavaScript les événements peuvent être traités par 3 mécanismes

1. Les fonctions de rappel (callbacks).
2. Les promesses (promises).
3. Le mécanisme async await.

Les fonctions de rappel (callbacks)

Une fonction de rappel (callback en anglais) est une fonction passée dans une autre fonction en tant qu'argument, qui est ensuite invoquée à l'intérieur de la fonction externe pour accomplir une action.

Voici un exemple de callback passée en paramètre de la fonction setTimeout.

```
const afficherTimeout = () => console.log('Timeout!');
setTimeout(afficherTimeout, 2000);
```

Ici, la fonction ‘afficherTimeout’ est invoquée au bout de 2000 millisecondes.

Les fonctions de rappel (callbacks)

L'inconvénient de l'approche par call back c'est quand on a un enchaînement d'événements. Dans ce cas, on peut avoir une imbrication de fonction call back à rappeler qui rend le code plus difficile à lire.

Dans l'exemple suivant, on appelle plusieurs fonctions asynchrones pour récupérer des commits d'un utilisateur.

```
getUserAsync(1, (user) => {
  getRepositoriesAsync(user.gitHubUsername, (repos) => {
    getCommitsAsync(repos[0], (commits) => {
      console.log('Commits', commits);
    });
  });
});
```

Les fonctions de rappel (callbacks)

L'autre inconvénient des fonctions callback c'est que la gestion des erreurs peut être lourde.
Prenons une fonction async '`calculateRacineAsync`' qui calcule une raciné carré. L'exemple suivant termine en erreur car le nombre passé en paramètre est négatif.

```
function calculateRacineAsync(nombre, callback) {
  setTimeout(() => {
    if (nombre < 0)
      throw new Error("nombre doit être positif");
    callback(Math.sqrt(nombre));
  }, 1000);
}
calculateRacineAsync(-1, result => console.log(`Résultat: ${result}`));
```

Même en mettant la fonction '`calculateRacineAsync`' dans un bloc try / catch l'erreur persiste car l'erreur de la fonction callback n'est pas gérée.

Les fonctions de rappel (callbacks)

Pour gérer l'erreur précédente de manière convenable, il faudrait ajouter un paramètre à la fonction callback. Ce paramètre contiendrait par exemple un message en cas d'erreur.

```
function calculateRacine(nombre, callback) {
  setTimeout(() => {
    if (nombre < 0)
      return callback(null, "nombre doit être positif");
    callback(Math.sqrt(nombre), null);
  }, 1000);
}
calculateRacine(-1, (result, error) => {
  if (error)
    return console.log(`Error: ${error}`);
  console.log(`Résultat: ${result}`);
});
```

Promises

Les promesses en JavaScript simplifient la gestion des événements.
Voici un exemple d'utilisation. Le résultat est récupéré à l'aide de la fonction 'then'.

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve(42), 2000);
});
promise.then(result => console.log(`Résultat: ${result}`));
```

Notez que le constructeur 'Promise' prend une callback ayant 2 arguments en paramètre. Ces 2 arguments sont des fonctions.

resolve: est appelée pour retourner le résultat de la promesse.
reject: est appelée en cas d'erreur.

Promises

L’exemple précédent de récupération des commits utilisateur, pourrait être simplifié comme suit

```
getUserAsync(1)
  .then(user => getRepositoriesAsync(user.gitHubUsername))
  .then(repos => getCommitsAsync(repos[0]))
  .then(commits => console.log('Commits', commits));
```

La fonction ‘then’ retourne une promesse. Ce qui nous permet d'enchaîner des promesses et d’éviter l’imbrication de blocs d’appel.

Promises

La gestion des erreurs est également simplifiée. La fonction ‘catch’ récupère les erreurs de la promesse.

```
function calculateRacine(nombre) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (nombre < 0)
        reject("nombre doit être positif");
      resolve(Math.sqrt(nombre));
    }, 1000);
  });
}
calculateRacine(-1)
  .then(result => console.log(`Résultat: ${result}`))
  .catch(error => console.log(`Erreur: ${error}`));
```

Async Await

Les fonctions async ne sont qu'un sucre syntaxique autour de la notion de promesse. Ils permettent l'utilisation d'une syntaxe proche de celle des fonctions synchrones.
Voici un exemple d'utilisation.

```
const getPromise = () => new Promise((resolve, reject) => {
  setTimeout(() => resolve(42), 2000);
});
async function maFonctionAsync() {
  const result = await getPromise();
  console.log(`Résultat: ${result}`);
}
maFonctionAsync();
```

Pour pouvoir utiliser le mot clé 'await', la fonction doit être déclarée à l'aide du mot clé 'async'.

Async Await

L'exemple précédent de récupération des commits utilisateur, pourrait être simplifié comme suit

```
const user = await getUser(1);
const repos = await getRepositories(user.gitHubUsername);
const commits = await getCommits(repos[0]);
console.log('Commits', commits);
```

Les fonctions 'getUser', 'getRepositories' et 'getCommits' doivent être déclarée à l'aide du mot clé 'async'.

La syntaxe est proche de celle des fonctions synchrones.

Async Await

La gestion des erreurs se fait à l'aide de blocs try / catch.
La fonction '`calculateRacine`' est la même que celle de l'exemple précédent

```
async function calculateRacineAsync(nombre) {
  try {
    const result = await calculateRacine(nombre);
    console.log(`Résultat: ${result}`);
  }
  catch(error) {
    console.log(`Erreur: ${error}`);
  }
}
calculateRacineAsync(-1);
```