Pourquoi faut-il des callback en Javascript ?[1]

Javascript propose la mise en place de callback, car le langage ne possède pas de support pour des exécution multithreadées, et que le mécanisme de callback est l'unique solution pour ne pas bloquer une exécution.
Explications : L'objectif de ce TD est de réaliser un certain nombre d'exercices afin de comprendre le principe du callback.

Faire une page html qui exécute un simple script contenant une fonction "coûteuse"

Il est possible qu'une fonction bloque de manière assez brutale une page html.

Exercice 1 Montrez, qu'à partir d'un certain "coût" de la fonction, la page à du mal à s'afficher.

```
<html>
  Hello world
  GoodBye World
  <script src="exercice1.js"></script>
</html>
```

```
function call1 () {
  console.log("i -->" + i);
}

for (i = 0; i < 200; i++) {
  call1();
 }
```

Conclusions ? Commentaires ?

Comprendre l'eventStack

La politique d'exécution d'une page html repose principalement sur un seul thread d'exécution. La fonction setTimeout, permet de placer dans la pile des prochaines fonctions à exécuter une fonction particulière. Javascript exécute cette pile fonction après fonction dans l'ordre de la pile. Par exemple :

Pile d'appel
main()
call1()

Javascript exécutera la f1, puis f2, puis f3. Après l'appel setTimeout(f4, 200, 3), javascript placera la fonction f4 après 200 ms à la suite des fonctions à exécuter

Pile t1	Pile t2	Pile t3	Pile t4
main()	call1(1)	call1()	...

===> Langage turn based

Exercice 2 Transformez le compteur précédent, dans une fonction strictement équivalente mais qui ne bloque pas le client.

L'arrivée du callback

On repart sur le pattern d'écriture de fonction non bloquante suivant :

```
//Pattern
function call1 () {
  function _call1(i) {
  //Ancienne fonction
    i++;
    console.log("i -->" + i);
    if (i < 2000) {
    //Pattern
      setTimeout(_call1, 0, i);
    }
   }
  _call1(0);
  return 'ok';
 }

 call1()
```

Comment récupérer la terminaison de la fonction call1 ?

Montrez ce qui se passe quand on traite la question de manière synchrone ?

Le pattern précédent ne bloque plus le client ! Certe, mais celui-ci ne sait alors plus quand la fonction se termine. Il est alors impossible d'écrire un programme, qui, à la fois utilise un code non-bloquant d'appel de fonction et qui bloque l'appelant le temps de son exécution.

Pour résoudre ce problème, on passe par le célèbre mécanisme de callback de Javascript. Qui est intimement lié au mécanisme de closure.

Un appel synchrone/bloquant s'écrit classiquement ainsi `ret = f3(25)`, bloque l'appelant le temps que f3 s'éxécute avec le paramètre `25`, et lui transmet le résultat dans la variable ret à la fin. Dans javascript les appels ne peuvent plus être bloquants, et la syntaxe précédent ne véhicule aucune information dans un certain nombre de cas.

Une convention à été choisie qui utilise le dernier paramètre d'une fonction asynchrone en tant que fonction que l'appelé utilisera pour notifier l'appelant de la fin de son exécution.

L'appel équivalent asynchrone devient : `f3(25, function() { console.log("Appel terminé");} )`

On remarquera l'utilisation d'une fonction anonyme comme dernier paramètre de l'appel, car oui, Javascript est un langage fonctionnel. C'est à dire que les paramètre d'une fonction peuvent être du type fonction. (Ce qui n'est pas possible dans les langages non fonctionnel comme Java par exemple).

Exercice 3 Transformez le code de la fonction coût pour notifier l'appelant à la fin de l'exécution. Montrez par la même occasion que l'appelant n'est pas bloqué dans son exécution, et que pourtant il ne 'quite pas' ; ce qui illustre le mécanisme de closure.

L'enchaînement des callbacks

L'utilisation des callback, permet donc de réaliser un programme sychrone non bloquant pour l'appelant, exécutant une suite d'action en fonction du résultat des actions précédentes alors que le contrôle global est fait de manière asynchrone non bloquante.

Exercice 4 Ecrivez le programme à base de callback équivalent à cet appel "classique". ret1 = appelBloquant1(); ret2 = appelBloquant2(); ret3 = appelBloquant3();

C'est à dire, formulé différemment, un programme garantissant que l'ordre d'appel des fonctions est conservé strictement. appelBloquant1 > appelBloquant2 > appelBloquant3, quelque soit le comportement d'attente des fonctions.

Gardez bien à l'esprit que les appelBloquants peuvent potentiellement coûter très chers, mais ne doivent pas perturber le navigateur

L'enfer des callbacks

Les callbacks génèrent une structure de lecture qui n'est pas alignée avec la structure d'exécution. Ceci est habituellement appelé l'enfer des callbacks.

Exercice 5 indiquez l'ordre d'apparition des messages le plus probable du code suivant.

```
console.log("A");
call1(function() {
  console.log("B");
  call2(function() {
    console.log("C");
    call3(function() {
      console.log("D");
    });
    console.log("E");
  });
  console.log("F");
});
console.log("G");
```

Il est clair que ce code, est moins clair que :

```
System.out.println("A");
ret1 = appelBloquant1();
System.out.println("B");
ret2 = appelBloquant2();
System.out.println("C");
ret3 = appelBloquant3();
```

Il est l'est d'autant plus que les codes de retour ne sont pour l'instant pas traités. Deux solutions sont alors possibles, soit continuer à faire évoluer le pattern, soit utiliser une bibliothèque offrant une vision simplifiée du patterns des callbacks sous le nom d'un paradygme commun.

Les promesses (implantée par Q, ou fibers()), Asynch, ou XXX sont de telles bibliothèques.

Exercice 6 Proposez un modèle standard de gestion des codes de retour.

---

1. © Stéphane Frénot, Département Télécommunications, cours ELP ↩