Programmation web
JavaScript - Langage 4 - Itérateurs et générateurs

Vincent Padovani, PPS, IRIF

1 Itérateurs

1.1 Introduction aux objets itérables

Informellement, un objet est itérable s'il est capable de fournir, à la demande et une par une, une certaine suite de valeurs. Les tableaux sont par exemple des objets itérables pouvant fournir la suite de leurs éléments. De même, les chaînes sont des objets itérables pouvant fournir la suite de leurs caractères.

Boucles `for`/`of` généralisées. Les boucles `for`/`of`, déjà présentées au Chapitre 1 pour le parcours de tableaux, permettent en fait d'examiner une à une la suite des valeurs fournie par tout objet itérable - non seulement les éléments d'un tableau, mais aussi les caractères d'une chaîne :

```
let sum = 0;
for (let i of [1, 3, 5]) { // pour chaque valeur i du tableau [1, 3, 5]
  sum += i;
}
sum;                       // => 9 === 1 + 3 + 5

let rev = "";
for (let c of "abc") {     // pour chaque caractere c de "abc"
  rev = c + rev
}
rev;                       // "cba"
```

Les objets ne sont pas tous itérables : une tentative de parcours par `for`/`of` d'un objet littéral ordinaire déclenchera par exemple une erreur d'exécution. Il existe par contre dans le constructeur prédéfini `Object` des méthodes utilitaires permettant de récupérer les données internes d'un objet sous forme de tableaux, qui peuvent évidemment être soumis à `for`/`of` :

- `Object.keys(o)` renvoie le tableau des noms de propriétés propres de `o` (mais pas ceux de ses propriétés héritées), dans l'ordre de leur ajout dans `o`,
- `Object.values(o)` renvoie le tableau des valeurs de ces propriétés,
- `Object.entries(o)` renvoie un tableau de tableaux à deux éléments, chaque tableau contenant un nom de propriété propre de `o` et sa valeur.

```
let o = {x : 1, y : 2, z : 3};
Object.keys(o);             // => ["x", "y", "z"]
Object.values(o);           // => [1, 2, 3]
Object.entries(o);          // => [["x", 1], ["y", 2], ["z", 3]]
```

Opérateur d'étalement. La suite des valeurs fournies par un objet itérable peut être explicitement manipulée à l'aide de l'*opérateur d'étalement* (spread operator), par exemple pour insérer cette suite dans celle d'un tableau littéral. Cet opérateur s'écrit sous la forme de trois points, mais ne doit pas être confondu avec la syntaxe de collecte des arguments restants (*c.f.* le Chapitre 1) :

```
let a = [1, 2, 3];
let b = [0, ...a, 4];  // b == [0, 1, 2, 3, 4]
let c = [..."abcde"];  // t == ["a", "b", "c", "d", "e"];
```

L'opérateur d'étalement permet par exemple de copier le contenu d'un tableau dans un autre avec une syntaxe minimaliste :

```
let t = [1, 2, 3];
let u = [...t];         // => [1, 2, 3]
t === u;                // => false // (valeurs de référence distinctes)
```

1.2 Itérables, itérateurs et résultats d'itération

La notion d'itération en JavaScript met en jeu trois catégories d'objets : les *itérables* (iterables), les *itérateurs* (iterators) et, les *résultats d'itération* (iteration results).

1. Un *résultat d'itération* est un objet muni d'une propriété `value` et/ou d'une propriété `done`. Un résultat est *valide* si sa propriété `value` est définie et si sa propriété `done` est indéfinie ou égale à `false`. Un résultat est une *fin d'itération* si sa propriété `done` est égale à `true`.
2. Un *itérateur* est un objet muni d'une méthode `next()` renvoyant un résultat d'itération. La suite de valeurs *énumérées* par cet itérateur est formée de toutes les valeurs encapsulées dans les résultats valides renvoyés par des invocations successives sa méthode `next()`, jusqu'à ce qu'elle renvoie pour la première fois une fin d'itération.
3. Un objet est *itérable* s'il est muni d'une méthode spéciale renvoyant un itérateur.

Le nom de la méthode permettant d'accéder à l'itérateur d'un itérable n'est pas standard : il s'agit de `Symbol.iterator`[1].

1. L'objet prédéfini `Symbol` est un élément utilitaire du langage permettant en particulier de créer de nouveaux *symboles*, c'est-à-dire des noms de propriétés uniques, jamais manipulables de manière directe, afin par exemple d'éviter de redéfinir accidentellement des propriétes d'objets prédéfinis dans des extensions du langage. Le symbole `Symbol.iterator` est un exemple de symbole prédéfini.

Méthode `next()`. Pour tout objet `o` itérable, `o[Symbol.iterator]()` renvoie un itérateur qui est celui utilisé par une boucle `for`/`of` ou par l'opérateur d'étalement : les valeurs considérées par l'un ou l'autre seront celles énumérées par cet itérateur. Il n'y a donc aucune différence opératoire entre le code suivant :

```
let sum = 0;
let iterable = [1, 3, 5];
for (let i of iterable) {
  sum += i;
}
```

et celui-ci :

```
let sum = 0
let iterable = [1; 2; 3; 4];
let iterator = iterable[Symbol.iterator]();
for (let res = iterator.next(); !res.done; res = iterator.next()) {
  sum += res.value
}
```

Comme indiqué plus haut, les objets ne sont pas en général itérables. Il est cependant possible d'ajouter explicitement à un objet une méthode de nom `Symbol.iterator` renvoyant un itérateur. L'objet devient de fait itérable après cet ajout, l'itérateur devenant celui qui sera utilisé par l'opérateur d'étalement ou par un `for`/`of`.

Dans l'exemple suivant, un constructeur permet de définir des objets représentant des intervalles. Un itérateur est associé à chaque intervalle créé, énumerant toutes les valeurs comprises entre ses bornes – c'est cet itérateur qui est utilisé par l'opérateur d'étalement à la dernière ligne :

```
function Range(min, max) {
  // ajout de deux champs min et max
  this.min = min;
  this.max = max;
  // ajout explicite d'un itérateur énumérant
  // toutes les valeurs entre min et max :
  this[Symbol.iterator] = function () {
    let n = min;
    return {
      next : function () {
        return (n <= max) ?
               { value: n++,       done : false } :
               { value: undefined, done : true  };
      }
    };
  }
}
let r = new Range(42, 52);
[...r];                     // => [42, 43, 44, 45, 46, 47, 48, 49, 50]
```

Méthode `return()`. Il est possible d'ajouter une seconde méthode à un itérateur de nom `return()` : la première fois que la que sa méthode `next()` renverra une fin d'itération, cette méthode sera invoquée avant le retour de ce résultat, par exemple pour effectuer une libération de ressources, une fin de connexion, une fermeture de fichier, etc.

2 Générateurs

Un *générateur* est un itérateur obtenu à partir d'une fonction de catégorie spéciale, une *fonction génératrice*. Une fonction génératrice se déclare comme une fonction ordinaire en ajoutant une étoile (`*`) entre le mot-clef `function` et le nom de cette fonction.

2.1 Méthode `next()` et instructions de productions

Une fonction génératrice ne s'exécute jamais de manière directe : un appel de cette fonction ne fait que renvoyer un générateur. C'est seulement lors de l'appel de la méthode `next()` de ce générateur que le corps de cette fonction génératrice démarrera son exécution à partir de sa premier instruction.

L'exécution du corps de la fonction génératrice se poursuit jusqu'à sa terminaison, ou jusqu'à la rencontre d'une *instruction de production*, une instruction de la forme `yield` *expr*. Dans le premier cas, le résultat de l'invocation de `next()` est une fin d'itération. Dans le second, ce résultat est valide et encapsule la valeur de l'expression qui suit le `yield` atteint. À l'invocation de `next()` suivante, l'exécution du corps de la fonction génératrice reprendra immédiatement après ce dernier `yield`.

Voici un exemple de fonction génératrice et de générateur. Noter qu'à la première invocation de `next()` l'instruction d'affichage n'est pas atteinte, puisqu'elle est précédé d'un `yield` qui la précède aussi dans l'ordre d'exécution :

```
// fonction génératrice
function* sequence(from, to) {
  for(let i = from; i <= to; i++) {
    yield i;
    console.log(i);
  }
}

// retour d'un générateur
let s = sequence(10, 13);
s.next().value;     // (i === from === 10) => 10
s.next().value;     // (affichage de 10, i++, yield i) => 11
s.next().value;     // (affichage de 11, i++, yield i) => 12
s.next().value;     // (affichage de 12, i++, yield i) => 13
s.next().done;      // (affichage de 13, i++, terminaison) => true
```

Noter qu'un générateur est aussi un itérable : il est son propre itérateur. On peut donc aussi écrire :

```
[...sequence(10, 13)]  // => [10, 11, 12, 13]
```

2.2 Générateurs infinis

Un itérateur peut ne renvoyer qu'un nombre fini de résultats valides avant de renvoyer une fin d'itération, mais cela n'a rien d'une obligation. Voici un exemple de fonction de génération produisant la suite infinie des multiples d'un nombre `n` donné :

```
function* multiples(n) {
    let m = 0;
    while(true) {
        yield m;
        m += n;
    }
}
let a = multiples(3);
a.next().value;        // 0
a.next().value;        // 3
a.next().value;        // 6
// etc.
```

Il est clair qu'un générateur produit par cette fonction – qui est aussi un itérable – ne saurait être soumis à l'opérateur de déploiement, sous peine de bouclage. Il est cependant possible de contrôler ce risque de bouclage à l'aide d'une boucle `for`/`of` volontairement interrompue au bout d'un nombre fini d'étapes.

Dans la fonction ci-dessous, une boucle `for`/`of` appliquée à `multiples(n)` – vu comme un itérable – est interrompue au bout de `p` étapes. Les `p` premiers multiples de `n` sont accumulés dans une liste renvoyée par la fonction :

```
function prefix(n, p) {
  let lm = [];
  for (m of multiples(n)) {
    if (p-- > 0) {
      lm.push(m);  // ajouter m à la fin de lm
      continue;
    }
    break;
  }
  return lm;
}

prefix(3, 5);      // => [0, 3, 6, 9, 12]
```