ななちゃんのIT教室

キーワードはイテラブルの巻

by nara.yasuhiro@gmail.com

ななちゃんが
イテラブルの概念を学ぶという お話

第 0.1 版　2017 年 6 月 8 日

フリー素材
http://freeillustration.net

いらすとやフリー素材
http://www.irasutoya.com/

もくじ

第1回　イテラブル（反復可能）とは

なな：「イテラブル」って何？

先生：「イテラブル」（iterable、反復可能）オブジェクトとは、「イテレータ」を生成出力するメソッドを、Symbol.iterator というプロパティ名の値として持つオブジェクトのことです。

なな：え？「イテレータ」って何？

先生：「イタレータ」（iterator、反復子）オブジェクトとは next() メソッドを持ち、それを繰り返し呼び出すことで、順次 {"value":"d1","done":false}、 {"value":"d2","done":false}、さいごに {"done":true} というようなオブジェクトを返すことで内容を取り出すことができるオブジェクトのことです。

なな：どちらも、今までに聞いたことが無いけど。

先生：ともに、JavaScript の新しい仕様である、EcmaScript 2015 から導入された概念です。

なな：2015？ ちょっと古い？

先生：仕様が発表されたのが 2015年ということね。 いろいろなブラウザで実際に使えるようになったのは、ごく最近のことなの。だから、勉強するにはちょうど良い「旬」ということかも。

なな：具体的には、JavaScript の何と関係があるの？

先生：String、Array、TypedArray、Map、Set といったビルトイン型のいくつかは、他の型（Object など）と違い、デフォルトでイテレータを出力する機能を備えています。こういうのを、ビルトイン反復可能オブジェクト（ビルトイン・イテラブル・オブジェクト）といいます。

なな：そういう機能は、どうやって使うの？

先生：イテラブルオブジェクトは、var it = “abc” [Symbol.iterator](); のような形でイテレータを生成することができます。そのほか、「for (v of iterable)」の形の「for-of 文」で使ったり、「Array.from(iterable)」のような形で配列に変換したり、「var m = new Map(iterable)」、「var s = new Set(iterable)」、「var a = new Array(iterable)」というような形で他のイテラブルオブジェクト生成に使ったり、「var array = [...iterable];」や、「func(...iterable)」のような展開演算子（…）とともに使ったり、「[a, b, c] = iterable」のような形で分割代入に利用したり、といったことができます。

なな：なんとなく、雰囲気は分かったような気がするけど、具体的なイメージがいまひとつのような。

先生：次回から、ひとつずつ、具体的に勉強しましょう。

第2回　イテレータとは

なな：「イテレータ」って、具体的には何？

先生：「イタレータ」（iterator、反復子）オブジェクトとは next() メソッドを持ち、それを繰り返し呼び出すことで、順次 {"value":"d1","done":false}、 {"value":"d2","done":false}、さいごに {"done":true} というようなオブジェクトを返すことで内容を取り出すことができるオブジェクトのことです。原理を知るために、手作業でイテレータをプログラミングしてみましょう。こういうのを、カスタムイテレータといいます。

```
function makeIterator(array) {
    var nextIndex = 0;
    return {
        next: function() {
            return nextIndex < array.length ?
                {value: array[nextIndex++], done: false} :
                {done: true};
        }
    };
}
```

なな：これがイテレータ？

先生：まだ、イテレータそのものではありません。これは、「イテレータ生成器」みたいなものです。これをつかって、イテレータを作ります。

```
var it; it = makeIterator(['d1', 'd2']);      // Object object {}
it.next()                                     // Object object {"value":"d1","done":false}
it.next()                                     // Object object {"value":"d2","done":false}
it.next()                                     // Object object {"done":true}
```

なな：たしかに、「順次 {"value":"d1","done":false}、 {"value":"d2","done":false}、さいごに {"done":true} というようなオブジェクトを返すことで内容を取り出すことができる」わね。 でも、「it.next()」 を何回も書くのはちょっとダサイというか。

先生：一般的な使い方は、下記のようになります。

```
var it; it = makeIterator(['d1', 'd2']);      // Object object {}
var d;                                        // Undefined undefined undefined
while(d=it.next(),d.done==false) log(d.value) // Undefined undefined undefined d1d2
```

なな：なるほど。

先生：復習すると、 JavaScript のイテレータは、一連の処理中の次の項目を返す next() メソッドを提供するオブジェクトということになります。このメソッドは done と value という 2 つのプロパティを持つオブジェクトを返す。 広く一般的には、イテレータとは、一連の処理中において現在の処理位置を把握しつつ、コレクション中の項目へ一つずつアクセスする方法を備えたオブジェクトのことです。

第3回　イテレータ生成器を簡単に作る「ジェネレータ」

なな：手作業で「イテレータ生成器」を作るのって、面倒そうね。

先生：そうね。カスタムイテレータを作るには、注意深いプログラミングが必要ね。そこで、JavaScript には、イテレータの実装を容易にする仕組みである「ジェネレータ」（generator）があります。

なな：ジェネレータ？

先生：ジェネレータは、イテレータ生成器として働く、特別な種類の関数（メソッド）です。。1つ以上の yield 式を持ち、function* 構文を使用している場合に、関数はジェネレータとなります。 2 つの書き方があります。

```
function* itGen() {
   var index = 0;
   while(true) yield index++;
}
itGen                                   // GeneratorFunction function undefined
```

```
var itGen; itGen = function*() {
   var index = 0;
   while(true) yield index++;
}                                  // GeneratorFunction function undefined
```

なな：function の書き方が 2 種類あるのと対応しているわね。どうやって使うの？

先生：このように使います。

```
itGen                              //GeneratorFunction function undefined
var it; it = itGen();              // Generator object {}
it.next()                          // Object object {"value":0,"done":false}
it.next()                          // Object object {"value":1,"done":false}
it.next()                          // Object object {"value":2,"done":false}
```

なな：このイテレータは、value がいくつまで出力されるの？

先生：これは、原理的なプログラムなので、1,2,3,4,5,,, と、無限に続いてしまいます。有限の例を掲げます。

```
var ItGen; itGen = function* () {
      yield 1;
      yield 2;
      yield 3;
};                                 // GeneratorFunction function undefined
```

```
var it; it = ItGen();              // Generator object {}
it.next()                          // Object object {"value":1,"done":false}
it.next()                          // Object object {"value":2,"done":false}
it.next()                          // Object object {"value":3,"done":false}
it.next()                          // Object object {"done":true}
```

先生：配列から順に取り出すこともできます。

```
function* gen() {
   yield* ['a', 'b', 'c'];
}
```

```
var g; g = gen() // Generator object {}
g.next()         // Object object {"value":"a","done":false}
g.next()         // Object object {"value":"b","done":false}
g.next()         // Object object {"value":"c","done":false}
g.next()         // Object object {"done":true}
```

なな：取り出すのをリセットするには？

先生：こうします。

```
function* strIter() {
      var str = "abc";
      var idx = 0;
      while(idx < str.length) {
            var modify = yield str[idx++];
            if(modify == 100) { idx = 0; }
      }
}
```

```
var si3; si3 = strIter()          // Generator object {}
si3.next()                              // Object object {"value":"a","done":false}
si3.next()                              // Object object {"value":"b","done":false}
si3.next()                              // Object object {"value":"c","done":false}
si3.next()                              // Object object {"done":false}
si3.next(100)                           // Object object {"value":"a","done":false}
si3.next()                              // Object object {"value":"b","done":false}
si3.next(5)                             // Object object {"value":"c","done":false}
si3.next(100)                           // Object object {"value":"a","done":false}
```

第4回　具体的な「イテラブル」オブジェクト

なな：ジェネレータでも、ちょっと面倒ね。

先生：JavaScript にはじめから用意されているビルトイン型のうち、String、Array、TypedArray、Map、Set などには、はじめからジェネレータの機能が内蔵されています。

なな：どんな形で？

先生：オブジェクトの Symbol.iterator をキーとするプロパティの値として、イテレータメ生成ソッド（@@iterator）を持っているの。

なな：どうやって利用するの？

先生：たとえば、文字列だったら、

```
var it; it = "abc"[Symbol.iterator]();        // String Iterator object {}
```

は、下記と同等です。

```
function* gen() { yield* "abc"; }
var it; it = gen()  // Generator object {}
```

下記のように使うことができます。

```
var it; it = "abc"[Symbol.iterator]();      // String Iterator object {}
it.next()                                   // Object object {"value":"a","done":false}
it.next()                                   // Object object {"value":"b","done":false}
it.next()                                   // Object object {"value":"c","done":false}
it.next()                                   // Object object {"done":true}

var it; it = "abc"[Symbol.iterator]();      // String Iterator object {}
for(var v of it) log(v+"/");                // Undefined undefined undefined a/b/c/

var it; it = "abc"[Symbol.iterator]();      // String Iterator object {}
[...it]                                     // Array object ["a","b","c"]
```

また、文字列自体でも、イテラブルになります。

```
for(var v of "abc") log(v+"/");     // Undefined undefined undefined a/b/c/
[..."abc"]                          // Array object ["a","b","c"]
```

配列についても同様です。また、配列の keys()、values()、entries() メソッドもイテレータを生成します

```
var it; it = [1,2,3][Symbol.iterator]();    // Array Iterator object {}
[...it]                                     // Array object [1,2,3]
var it; it = [1,2,3].keys();                // Array Iterator object {}
[...it]                                     // Array object [0,1,2]
var it; it = [1,2,3].values();              // Array Iterator object {}
[...it]                                     // Array object [1,2,3]
var it; it = [1,2,3].entries();             // Array Iterator object {}
[...it]                                     // Array object [[0,1],[1,2],[2,3]]
for(var v of [1,2,3]) log(v+"/");           // Undefined undefined undefined 1/2/3/
[...[1,2,3]]                                // Array object [1,2,3]
```

イテレータ は、それ自身がイテラブルなオブジェクト。ゆえに、[Symbol.iterator]() メソッドを実行すると、自分自身を返します。

```
var it; it = [1,2,3][Symbol.iterator]();      //Array Iterator object {}
it === it[Symbol.iterator]();                 // Boolean boolean true
```

ジェネレータ関数から生成される イテレータも、イテラブルなオブジェクトです。

```
function* gfn(n){
      while(n < 100){
            yield n;
            n *= 2;
      }
}
var it; it = gfn(24);                 // Generator object {}
for(var v of it) log(v+"/");          // Undefined undefined undefined 24/48/96/

var it; it = gfn(24);                 // Generator object {}
[...it]                               // Array object [24,48,96]
```

その他にも for～of や [...～] の対象となるイテラブルなオブジェクトがいろいろあります。

＜引数＞

```
--------------------------------------------------------------------------------
function func(){
      for(var v of arguments) log(v+"/");
}
func(1,2,3);                                  // Undefined undefined undefined
--------------------------------------------------------------------------------
```

＜TypedArray＞

```
--------------------------------------------------------------------------------
var view; view = new Uint8Array([0, 1, -1]); // Uint8Array object {"0":0,"1":1,"2":255}
for(var v of view) log(v+"/");                 // Undefined undefined undefined 0/1/255/
--------------------------------------------------------------------------------
```

＜Map＞

```
--------------------------------------------------------------------------------
var map; map = new Map([[0, "Zero"], [{}, "Object"], [[], "Array"]]); // Map object {}
[...map]                                      // Array object [[0,"Zero"],[{},"Object"],[[],"Array"]]
--------------------------------------------------------------------------------
```

＜Set＞

```
--------------------------------------------------------------------------------
var set; set = new Set([0, {}, []]);      // Set object {}
[...set]                                          // Array object [0,{},[]]
--------------------------------------------------------------------------------
```

先生：イテラブルオブジェクトは、自分で作ることもできます。

```
var myIterable; myIterable = {};// Object object {}

myIterable[Symbol.iterator] = function () {
      var array = [1,2,3];
      var nextIndex = 0;
      return {
          next: function() {
                return nextIndex < array.length ?
                      {value: array[nextIndex++], done: false} :
                      {done: true};
          }
      };
};                          // Function function undefined

myIterable                  // Object object {}

var it; it = myIterable[Symbol.iterator]();          // Object object {}
it.next();                  // Object object {"value":1,"done":false}

[...myIterable]             // Array object [1,2,3]

var it; it = myIterable[Symbol.iterator]();          // Object object {}
[...it]                                              // ! TypeError: Function expected
```

普通の Object

生成するイテレータは Symbol.iterator 属性を持たない

```
var myIterable; myIterable = {};                     // Object object {}

myIterable[Symbol.iterator] = function* () {
    yield* [1,2,3];
};                                                   // GeneratorFunction function undefined

myIterable                                           // Object object {}

var it; it = myIterable[Symbol.iterator]();          // Generator object {}
it.next();                                           // Object object {"value":1,"done":false}

[...myIterable]                                      // Array object [1,2,3]

var it; it = myIterable[Symbol.iterator]();          // Generator object {}
[...it]                                              // Array object [1,2,3]
```

Generator

function*で生成するイテレータ（generator）は Symbol.iterator 属性を持つ

第5回　まとめ

先生：イテレータ の利用法をまとめておきましょう。

・ [...iterable] という構文。

iterable から順番に値が取り出されて、個数分の要素が該当部分に入るような配列を作成できます。

```
--------------------------------------------------------------------------
var ary; ary = [0, "A", false];          // Array object [0,"A",false]
[...ary]                                 // Array object [0,"A",false]
--------------------------------------------------------------------------
var str; str = "ABC"                     // String string "ABC"
[...str]                                 // Array object ["A","B","C"]
--------------------------------------------------------------------------
var ary; ary = [0, "A", false];          // Array object [0,"A",false]
var str; str = "あいう";                 // String string "あいう"
var conAry; conAry = [...ary, ...str];   // Array object [0,"A",false,"あ","い","う"]
--------------------------------------------------------------------------
```

・ 引数渡し

```
--------------------------------------------------------------------------
var nums; nums = [65, 66, 67];           // Array object [65,66,67]
Math.max(...nums);                       // Number number 67
String.fromCharCode(...nums);            // String string "ABC"
--------------------------------------------------------------------------
```

・ 分割代入

[a, b, c] = iterable という構文。iterable から順に値が取り出され、左辺の変数に順に代入されます。

```
--------------------------------------------------------------------------
var [a, b, c] = "XYZ";                   // String string "XYZ"
c+b+a                                    // String string "ZYX"
--------------------------------------------------------------------------
```

・ Map, Set

new Map(iterable), new Set(iterable) という構文。
それぞれ iterable からは順番に値が取り出されて、キーや値を指定することができます。

```
--------------------------------------------------------------------------
var set; set = new Set("あいうあお");            // Set object {}
Array.from(set)                                  // Array object ["あ","い","う","お"]
--------------------------------------------------------------------------
var it; it = ["A", "B", "C"].entries()           // Array Iterator object {}
Array.from(it)                                   // Array object [[0,"A"],[1,"B"],[2,"C"]]
var map; map = new Map(["A", "B", "C"].entries()); // Map object {}
Array.from(map);                                 // Array object [[0,"A"],[1,"B"],[2,"C"]]
--------------------------------------------------------------------------
```

先生：イテラブルなオブジェクト の種類をまとめておきましょう。

自分で作る	略
配列	["A", "B", "C"]
配列の.keys()	["A", "B", "C"].keys()
配列の.values()	["A", "B", "C"].values()
配列の.entries()	["A", "B", "C"].entries()
文字列	"あいう"
イテレータ	略
ジェネレータ	(function*(){})()
Arguments	(function(){arguments/*←コレ*/})()
TypedArray	new Uint8Array([0, 1, -1])
Map	new Map()
Set	new Set()

先生：イテラブルなオブジェクト の利用法も、まとめておきましょう。

for-of 文	for(v of iterable)
配列リテラル	[...iterable]
Array.from	Array.from(iterable)
引数渡し	func(...iterable)
分割代入	[a, b, c] = iterable
Array	new Array(iterable)
Map	new Map(iterable)
Set	new Set(iterable)
yield*	yield* iterable

```
---------------------------------- 例 ------------------------------------
var x, y, z;
var s; s = new Set(['a', 'b', 'c']); // Set object {}
[x,y,z] = s;                         // Set object {}
x                                            // String string "a"
y                                            // String string "b"
z                                            // String string "c"
--------------------------------------------------------------------------
```

先生：応用例です。

配列をコピーする

```
var ary0 = [1, 2, 3];
var ary1 = [...ary0];
ary0.join() === ary1.join();     // true
ary0 === ary1;                           // false
```

配列の最初の要素を代入する

```
var ary = ["A", "B", "C"];
var [first] = ary;
first;                               // "A"
```

文字列の最初の文字を代入する

```
var str = "ABC";
var [first] = str;
first; // "A"
```

配列から重複した値を削除する

```
var ary = [0, 5, 9, 0, 2, 5];
var uniqueAry = [...new Set(ary)];
uniqueAry;                          // [ 0, 5, 9, 2 ]
```

.apply()を使わずに関数に可変長の引数を渡す

```
var nums = [112, 105, 121, 111];
Math.max(...nums);              // 121
String.fromCharCode(...nums)        // "piyo"
```

マッチした文字と、部分マッチした文字を一気に代入する

```
var [all, part] = "abcde".match(/ab(.)de/)
all;                        // "abcde"
part;                       // "c"
```

先生：さいごに、フィボナッチ数の例です。

```
---------------------------------------------------------------------------
function* fibonacci() {
   var fn1 = 0;
   var fn2 = 1;
   while (true) {
      var current = fn1;
      fn1 = fn2;
      fn2 = current + fn1;
      var reset = yield current;
      if (reset) {
            fn1 = 0;
            fn2 = 1;
      }
   }
}
var s; s = fibonacci();   // Generator object {}
s.next()                  // Object object {"value":0,"done":false}
s.next()                  // Object object {"value":1,"done":false}
s.next()                  // Object object {"value":1,"done":false}
s.next()                  // Object object {"value":2,"done":false}
s.next()                  // Object object {"value":3,"done":false}
s.next()                  // Object object {"value":5,"done":false}
s.next()                  // Object object {"value":8,"done":false}
s.next()                  // Object object {"value":13,"done":false}
s.next(true)              // Object object {"value":0,"done":false}
s.next()                  // Object object {"value":1,"done":false}
s.next()                  // Object object {"value":1,"done":false}
s.next()                  // Object object {"value":2,"done":false}
---------------------------------------------------------------------------
```