Пользовательские итераторы

В дополнение к стандартным встроенным итераторам теперь вы можете создать собственный. Для обеспечения взаимодействия с элементами ES6, использующими итераторы (например, с циклом `for..of` или оператором `...`), достаточно корректного интерфейса (или интерфейсов).

Давайте создадим итератор, генерирующий бесконечную последовательность чисел Фибоначчи:

```
var    Fib = {
    [Symbol.iterator]() {
        var    n1 = 1, n2 = 1;

        return {
            // делаем итератор итерируемым
            [Symbol.iterator]() { return this; },

            next() {
                var current = n2;
                n2 = n1;
                n1 = n1 + current;
                return { value: current, done: false };
            },

            return(v) {
                console.log(
                    "Последовательность Фибоначчи завершена."
                );
                return { value: v, done: true };
            }
        };
    }
};

for    (var v of Fib) {
    console.log( v );

    if    (v > 50) break;
}
// 1 1 2 3 5 8 13 21 34 55
// Последовательность Фибоначчи завершена
```

Без оператора `break` цикл `for..of` работал бы бесконечно, а это не имеет смысла.

Метод `Fib[Symbol.iterator]()` возвращает объект-итератор, обладающий методами `next()` и `return(..)`. Состояние поддерживается посредством переменных `n1` и `n2`, сохраненных замыканием.

А теперь рассмотрим итератор, выполняющий по очереди некоторые действия:

```
var    tasks = {
    [Symbol.iterator]() {
        var steps = this.actions.slice();

        return {
            // делаем итератор итерируемым
            [Symbol.iterator]() { return this; },

            next(...args) {
                if    (steps.length > 0) {
                    let res = steps.shift()( ...args );
                    return { value: res, done: false };
                }
                else {
                    return { done: true }
                }
            },

            return(v) {
                steps.length = 0;
                return { value: v, done: true };
            }
        };
    },
    actions: []
};
```

Итератор объекта `tasks` перебирает функции, обнаруженные в свойстве-массиве `actions`, и выполняет по очереди, передавая в них аргументы, которые ранее были переданы методу `next(..)`,

а затем возвращая полученное значение в виде стандартного объекта `IteratorResult`.

Вот как мы можем воспользоваться этой очередью задач:

```
tasks.actions.push(
    function step1(x){
        console.log( "step 1:", x );
        return x * 2;
    },
    function step2(x,y){
        console.log( "step 2:", x, y );
        return x + (y * 2);
    },
    function step3(x,y,z){
        console.log( "step 3:", x, y, z );
        return (x * y) + z;
    }
);

var it = tasks[Symbol.iterator]();

it.next( 10 );                // step 1: 10
                              // { value: 20, done: false }

it.next( 20, 50 );            // step 2: 20 50
                              // { value: 120, done: false }

it.next( 20, 50, 120 );       // step 3: 20 50 120
                              // { value: 1120, done: false }

it.next();                    // { done: true }
```

Пример демонстрирует, что итераторы могут применяться в качестве шаблона не только для данных, но и для структурирующих алгоритмов. Мы еще поговорим об этом в следующем разделе при обсуждении генераторов.

Можно даже подойти к вопросу творчески и задать итератор, выполняющий метаоперации с одним фрагментом данных. Например, давайте определим итератор для чисел в диапазоне

от 0 до какого-то заданного положительного или отрицательного значения:

```
if    (!Number.prototype[Symbol.iterator]) {
    Object.defineProperty(
        Number.prototype,
        Symbol.iterator,
        {
            writable: true,
            configurable: true,
            enumerable: false,
            value: function iterator(){
                var i, inc, done = false, top = +this;

                // итерации в положительную или отрицательную
                // сторону?
                inc = 1 * (top < 0 ? -1 : 1);

                return {
                    // делаем итерируемым сам итератор
                    [Symbol.iterator](){ return this; },

                    next() {
                        if    (!done) {
                            // начальная итерация всегда 0
                            if    (i == null) {
                                i = 0;
                            }
                            // итерации в положительном
                            // направлении
                            else if (top >= 0) {
                                i = Math.min(top,i + inc);
                            }
                            // итерации в отрицательном
                            // направлении
                            else {
                                i = Math.max(top,i + inc);
                            }

                            // закончить после этой итерации?
                            if    (i == top) done = true;

                            return { value: i, done: false };
```

```
                        }
                        else {
                            return { done: true };
                        }
                    }
                };
            }
        }
    );
}
```

Что же мы получили в результате такого творческого подхода?

```
for    (var i of 3) {
    console.log( i );
}
// 0 1 2 3

[...-3];                // [0,-1,-2,-3]
```

Эти приемы кажутся забавными, хотя их практическая польза остается до некоторой степени спорной. В то же время может возникнуть вопрос: почему в ES6 отсутствует эта незначительная, но приятная функциональная особенность?

Было бы упущением не напомнить вам, что к расширению встроенных прототипов, пример одного из которых вы видели в последнем фрагменте кода, нужно подходить аккуратно и все время помнить о потенциальных опасностях.

В рассмотренном случае шансы конфликта с другим кодом или с какой-нибудь будущей функциональной особенностью JS, скорее всего, исчезающее малы. Но даже к такой малой вероятности следует подходить серьезно, поэтому подробно документируйте такие вещи, чтобы избежать проблем в будущем.

Я расширил эту технику, см. http://blog.getify.com/iterating-es6-numbers/. В одном из комментариев к записи http://blog.getify.com/iterating-es6-numbers/comment-page-1/#comment-535294 мне предложили использовать аналогичный прием для диапазонов строковых символов.

Применение итераторов

Вы уже видели, как итератор шаг за шагом работает в цикле `for..of`. Но им могут пользоваться и другие структуры.

Рассмотрим итератор, присоединенный к массиву (хотя аналогичное поведение будет демонстрировать любой выбранный нами итератор):

```
var a = [1,2,3,4,5];
```

Оператор разделения `...` исчерпывает итератор полностью. Например:

```
function foo(x,y,z,w,p) {
    console.log( x + y + z + w + p );
}

foo( ...a );                    // 15
```

Кроме того, можно распределить итератор внутри массива:

```
var b = [ 0, ...a, 6 ];
b;                              // [0,1,2,3,4,5,6]
```

При процедуре деструктуризации массива (см. раздел «Деструктурирующее присваивание» главы 2) итератор может использоваться как частично, так и полностью (в сочетании с rest/gather-оператором `...`).

```
var it = a[Symbol.iterator]();

var [x,y] = it;
// берем из 'it' только первые два элемента
var [z, ...w] = it;
// берем третий элемент, а затем сразу все остальные

// истощен ли 'it' полностью? Да
it.next();            // { value: undefined, done: true }

x;                    // 1
```