JSDG 🦏 | Iterators and Generators | Итераторы и генераторы | chapter 12

`JavaScript: The Definitive Guide 7th EDITION • Master the World's Most-Used Programming Language • David Flanagan • 2021`

___

Особенности работы итераторов

Существуют три отдельных типа, которые необходимо освоить для понимания итерации в JavaScript.

Во-первых, есть *итерируемые* объекты: типы наподобие Array, Set и Map, по которым можно организовать итерацию.
Во-вторых, имеется сам объект *итератор* который выполняет итерацию.
И, в-третьих, есть объект *результата итерации*, который хранит результат каждого шага итерации.

Итерируемый объект — это любой объект со специальным итераторным методом, который возвращает объект итератора.
Итератор — это любой **объект** с методом next(), который возвращает объект результата итерации.

А объект результата *итерации* —- это объект со свойствами по имени value и done.
Для выполнения итерации по итерируемому объекту вы сначала вызываете его итераторный метод, чтобы получить объект итератора.
Затем вы многократно вызываете метод next () объекта итератора до тех пор, пока свойство done возвращенного значения не окажется установленным в true.

Сложность здесь в том, что итераторный метод итерируемого объекта не имеет обыкновенного имени, а взамен использует символьное имя Symbol.iterator.

```
let iterable = [991;
let iterator = iterable [Symbol.iterator] ();
for (let result = iterator. next () ; !result. done; result = iterator.next()) {
      console.log (result.value) // result.value == 99
```

}

объект итератора встроенных итерируемых типов данных сам является итерируемым.

(То есть он имеет метод с символьным именем symbol. iterator, который просто возвращает сам объект.)

Иногда это полезно в коде следующего кода, когда вы хотите выполнить проход по "частично использованному" итератору:

```
let list = [1,2,3,4, 5];
let iter = list [Symbol.iterator] () ;

let head = iter.next ().value; / head == 1

let tail = [...iter]; / tail == (2,3, 4, 5]
```

——

Реализация итерируемых объектов

Чтобы сделать класс итерируемым, потребуется реализовать метод с произвольным именем Symbol.iterator,
который должен возвращать объект итератора, имеющий метод n e x t ().

А метод n e x t () обязан возвращать объект результата итерации, который имеет свойство value и/или булевское свойство done.

```
*
* Объект Range представляет диапазон чисел (x: from <= x <= to).
* В классе Range определен метод has () для проверки,
* входит ли заданное число в диапазон.
* Класс Range итерируемый и обеспечивает проход по всем
целым
* числам внутри диапазона.
*/

class Range /
      constructor (from, to) 1
            this. from = from;
            this.to = to;
```

```
    }

    // Сделать класс Range работающим подобно множеству Set
чисел.
    has (x) { return typeof x === "number" && this.from <= x && x <=
this.to; }

    // Возвратить строковое представление диапазона, используя
запись множества.
    toString () { return `{ x | $(this. from) ≤ x ≤ $(this.to}; `;}

    // Сделать класс Range итерируемым за счет возвращения
объекта итератора.
    // Обратите внимание на то, что именем этого метода
является
    // специальный символ, а не строка.

    [Symbol.iterator] () {

    // Каждый экземпляр итератора обязан проходить по
диапазону
    // независимо от других. Таким образом, нам нужна
переменная
    // состояния, чтобы отслеживать местоположение в
итерации.
    // Мы начинаем с первого целого числа, которое больше или
равно from.

    let next = Math.ceil (this.from); // Это значение мы возвращаем
следующим.
    let last = this.to; // Мы не возвращаем ничего, что больше
этого.

    return { // Это объект итератора.
        // Именно данный метод next () делает это объектом
итератора.
        // Он обязан возвращать объект результата итерации.

        next () {
            return (next <= last) // Если пока не возвратили
последнее
                ? { value: next++ } // значение, возвратить
```

```
следующее значение и инкрементировать его,
                    : { done: true }; // в противном случае указать, что
все закончено.
                },

                // Для удобства мы делаем сам итератор
итерируемым.

                [Symbol.iterator] () { return this; }
            };
        }
}

for (let x of new Range (1, 10)) console. log (x) ; / / Выводятся числа
от 1 до 10
[...new Range (-2,2)] //  [-2, -1, 0, 1, 2]

// Возвращает итерируемый объект, который проходит по
результату
// применения f () к каждому значению из исходного
итерируемого объекта.

function map (iterable, f) {
    let iterator = iterable [Symbol.iterator] ();
    return { // Этот объект является и итератором, и
итерируемым.

        [Symbol.iterator] () { return this; },

        next () {
            let v = iterator.next () :

            if (v.done) {
                return v;
            } else {
                return { value: f (v.value) };
            }
        }
    }
}

//Отобразить диапазон целых чисел на их квадраты и
```

преобразовать в массив.
[.. map (new Range (1, 4), x => x*x)] // => [1, 4, 9, 16] [1, 4, 9, 16]

```
// Возвращает итерируемый объект, который фильтрует
указанный
// итерируемый объект, проходя только по тем элементам,
// для которых предикат возвращает true.

function filter (iterable, predicate) {
      let iterator = iterable [Symbol.iterator] ();
      return { // Этот объект является и игератором, и
итерируемым.
            [Symbol.iterator] () { return this; ],

            next () {
                  for (;;) {
                        let v = iterator.next () ;

                        if (v.done ll predicate(v.value)) (
                        return v;
                  }
            }
}     ;
}
// Отфильтровать диапазон, чтобы остались только четные числа.
[...filter (new Range (1,10), × => × & 2 === 0)] // => (2,4, 6,8, 10]
```

Закрытие" итератора: метод return()

Но итераторы не всегда работают до конца: цикл f o r / o f может быть пре кращен посредством break или return либо из-за исключения.
Аналогично, когда итератор используется с деструктурирующей операцией, метод next () вызывается лишь столько раз, сколько достаточно для получения значений для всех указанных переменных. Итератор может иметь гораздо больше значений, которые он способен возвратить, но они никогда не будут запрошены.

—

Генераторы

Генератор - это своего рода итератор, определенный с помощью нового мощного синтаксиса ES6; он особенно удобен, когда значения, по которым нужно выполнять итерацию, являются не элементами какой-то структуры данных, а результатом вычисления.

Для создания генератора сначала потребуется определить генераторную функцию. Генераторная функция синтаксически похожа на обыкновенную функцию JavaScript, но определяется с помощью ключевого слова function* вместо function.

(Формально это не новое ключевое слово, а просто символ * после ключевого слова function и перед именем функции.)

При вызове генераторной функции тело функции фактически не выполняется, но взамен возвращается объект генератора, который является итератором.

Вызов его метода next () за ставляет тело генераторной функции выполняться с самого начала (или с любой текущей позиции), пока не будет достигнут оператор yield.

Оператор yield появился в ES6 и кое в чем похож на оператор return.

Значение оператора yield становится значением, которое возвращается вызовом метода next () итератора.

Вот пример, который должен все прояснить:

```
// Генераторная функция, которая выдает набор простых чисел
// с одной цифрой (с основанием 10).

function* oneDigitPrimes () { // При вызове этой функции код
    //не выполняется, а просто
    yield 2; // возвращается объект генератора.
    // Вызов метода next ()
    yield 3; // данного генератора приводит
    // к выолнению кода до тех пор,
```

```
    yield 5; // пока оператор yield не предоставит
    // возвращаемое значение
    yield 7; // для метода next () .
}

// Когда мы вызываем генераторную функцию, то получаем
генератор.
let primes = oneDigitPrimes () ;

// Генератор - это объект итератора, который проходит
// по выдаваемым значениям.

primes.next () .value // => 2
primes.next () .value => 3
primes.next () .value ¡ => 5
primes.next () . value ¡ => 7
primes.next () . done [/ => true

// Генераторы имеют метод Symbol.iterator, что делает их
итерируемыми.
primes [Symbol.iterator] () // => primes
// Мы можем использовать генераторы подобно другим
итерируемым типам.

[..oneDigitPrimes()] // => [2,3,5,71

let sum = 0;
for (let prime of oneDigitPrimes ()) sum += prime;
sum // => 17

const seq = function* (from, to) {
    for (let i = from; i <= to; i++) yield i;
};
[... seq (3,5)] // => (3, 4, 51



let o = {
    x: 1, y: 2, z: 3,
    // Генератор, который выдает каждый ключ этого объекта.
    *g() {
        for (let key of Object. keys (this))  yield key;
    }
```

```
};

[...o.g()) /1 => ["*", "y", "z", "g"]
```

Обратите внимание, что записать генераторную функцию с применением синтаксиса стрелочных функций не удастся.

Генераторы часто упрощают определение итерируемых классов. Мы можем заменить метод [Symbol.iterator] (), представленный в примере 12.1, намно го более короткой генераторной функцией *[Symbol.iteratorsrbrack; O, которая выглядит следующим образом:

```
* [Symbol.iterator] () {
      for (let x = Math.ceil (this.from); x <= this.to; x++) yield x;
}
```

___

Примеры генераторов

```
function* fibonacciSequence () {
      let x = 0, y = 1;

      for (;;) {
            yield y;
            [x, y] = [y, x+y]: // Примечание: деструктурирующее
присваивание.
      }



}

// возвращает n-ное число фибоначчи.
function fibonacci (n) {
      for (let f of fibonacciSequence ()) {
            if (n-- <= 0) return f;
      }
}

fibonacci(20) //  10946
```

```
// Выдает первые п элементов указанного итерируемого объекта.

function* take (n, iterable) 1
    let it = iterable [Symbol.iterator) () ; / / Получить итератор для итерируемого объекта.

    while (n-- > 0) { // Цикл п раз:
        let next = it.next (); // Получить следующий элемент из итератора.

        if (next .done) return; // Если больше нет значений, тогда выполнить возврат раньше,

        else yield next. value; / / иначе выдать значение.
    }
}

// Массив из первых пяти чисел фибоначчи.
[... take (5, fibonacciSequence () ) ] // => [1, 1, 2, 3, 51]
```

___

yield * и рекурсивные генераторы

Ключевое слово yield* похоже на yield, но вместо выдачи одиночного значения оно проходит по итерируемому объекту и выдает каждое результирующее значение

```
function* sequence (...iterables) {
    for (let iterable of iterables) !
        yield* iterable;
    }
}

[... sequence ("abc", oneDigitPrimes()] // => ["a", "b", "c", 2,3,5, 7]
```

___

Генераторы — очень мощная обобщенная структура управления. Они дают нам возможность приостанавливать вычисление с

помощью y i e l d и снова пере. запускать его в произвольно более позднее время с произвольным входным значением.

Генераторы можно применять для создания своего рода кооперативной потоковой системы внутри однопоточного кода JavaScript.

Кроме того, генераторы можно использовать для маскировки асинхронных частей программы, чтобы код выглядел последовательным и синхронным, даже если некоторые вызовы функций на самом деле являются асинхронными и зависят от событий из сети.
