MDN

> Эта страница была переведена с английского языка силами сообщества. Вы тоже можете внести свой вклад, присоединившись к русскоязычному сообществу MDN Web Docs.

# Итераторы и генераторы

Обработка каждого элемента коллекции является весьма распространённой операцией. JavaScript предоставляет несколько способов перебора коллекции, от простого цикла `for` до `map()`, `filter()` и `array comprehensions (en-US)`. Итераторы и генераторы внедряют концепцию перебора непосредственно в ядро языка и обеспечивают механизм настройки поведения `for...of` циклов.

Подробнее см. также:

- `Iteration protocols`
- `for...of`
- `function*` и `Generator`
- `yield` и `yield*`
- `Generator comprehensions (en-US)`

## Итераторы

Объект является итератором, если он умеет обращаться к элементам коллекции по одному за раз, при этом отслеживая своё текущее положение внутри этой последовательности. В JavaScript итератор - это объект, который предоставляет метод next(), возвращающий следующий элемент последовательности. Этот метод возвращает объект с двумя свойствами: done и value.

После создания, объект-итератор может быть явно использован, с помощью вызовов метода next().

JS

```
function makeIterator(array) {
  var nextIndex = 0;

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

После инициализации, метод next() может быть вызван для поочерёдного доступа к парам ключ-значение в объекте:

JS

```
var it = makeIterator(["yo", "ya"]);
console.log(it.next().value); // 'yo'
console.log(it.next().value); // 'ya'
console.log(it.next().done); // true
```

## Генераторы

В то время как пользовательские итераторы могут быть весьма полезны, при их программировании требуется уделять серьёзное внимание поддержке внутреннего состояния. Генераторы предоставляют мощную альтернативу: они позволяют определить алгоритм перебора, написав единственную функцию, которая умеет поддерживать собственное состояние.

Генераторы - это специальный тип функции, который работает как фабрика итераторов. Функция становится генератором, если содержит один или более `yield` операторов и использует `function*` синтаксис.

JS

```
function* idMaker() {
  var index = 0;
  while (true) yield index++;
}

var it = idMaker();

console.log(it.next().value); // 0
console.log(it.next().value); // 1
console.log(it.next().value); // 2
// ...
```

## Итерируемые объекты

Объект является итерируемым, если в нем определён способ перебора значений, то есть, например, как значения перебираются в конструкции `for..of`. Некоторые встроенные типы, такие как `Array` или `Map (en-US)`, по умолчанию являются итерируемыми, в то время как другие типы, как, например, `Object`, таковыми не являются.

Чтобы быть итерируемым, объект обязан реализовать метод `@@iterator`, что означает, что он (или один из объектов выше по цепочке прототипов) обязан иметь свойство с именем `Symbol.iterator`:

### Пользовательские итерируемые объекты

Мы можем создать свои собственные итерируемые объекты так:

JS

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

### Встроенные итерируемые объекты

Объекты `String` , `Array` , `TypedArray` , `Map (en-US)` и `Set` являются итерируемыми, потому что их прототипы содержат метод `Symbol.iterator` .

## Синтаксис для итерируемых объектов

Некоторые выражения работают с итерируемыми объектами, например, `for-of` циклы, `spread operator`, `yield*` , и `destructuring assignment`.

JS

```
for (let value of ["a", "b", "c"]) {
  console.log(value);
}
// "a"
// "b"
// "c"

[..."abc"]; // ["a", "b", "c"]

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

gen().next()[(a, b, c)] = // { value:"a", done:false }
  new Set(["a", "b", "c"]);
a; // "a"
```

## Продвинутые генераторы

Генераторы вычисляют результаты своих yield выражений по требованию, что позволяет им эффективно работать с последовательностями с высокой вычислительной сложностью, или даже с бесконечными последовательностями, как продемонстрировано выше.

`Метод` `next()` также принимает значение, которое может использоваться для изменения внутреннего состояния генератора. Значение, переданное в next(), будет рассматриваться как результат последнего yield выражения, которое приостановило генератор.

Вот генератор чисел Фибоначчи, использующий `next(x)` для перезапуска последовательности:

JS

```
function* fibonacci() {
  var fn1 = 1;
  var fn2 = 1;
  while (true) {
    var current = fn2;
    fn2 = fn1;
    fn1 = fn1 + current;
    var reset = yield current;
    if (reset) {
      fn1 = 1;
      fn2 = 1;
    }
  }
}

var sequence = fibonacci();
console.log(sequence.next().value); // 1
console.log(sequence.next().value); // 1
console.log(sequence.next().value); // 2
console.log(sequence.next().value); // 3
console.log(sequence.next().value); // 5
console.log(sequence.next().value); // 8
console.log(sequence.next().value); // 13
console.log(sequence.next(true).value); // 1
console.log(sequence.next().value); // 1
console.log(sequence.next().value); // 2
console.log(sequence.next().value); // 3
```

> Примечание: Интересно, что вызов `next(undefined)` равносилен вызову `next()`. При этом вызов next() для нового генератора с любым аргументом, кроме undefined, спровоцирует исключение `TypeError`.

Можно заставить генератор выбросить исключение, вызвав его метод `throw()` и передав в качестве параметра значение исключения, которое должно быть выброшено. Это исключение будет выброшено из текущего приостановленного

контекста генератора так, будто текущий приостановленный `yield` оператор являлся `throw` оператором.

Если `yield` оператор не встречается во время обработки выброшенного исключения, то исключение передаётся выше через вызов `throw()`, и результатом последующих вызовов `next()` будет свойство `done` равное `true`.

У генераторов есть метод `return(value)`, который возвращает заданное значение и останавливает работу генератора.