Содержание

ГЛАВА 12

Итераторы и генераторы

Итерируемые объекты и ассоциированные с ними итераторы являются средством ES6, которое используется в книге повсеместно. Массивы (в том числе типизированные) итерируемы, в равной степени как строки и объекты `Set` и `Map`. Это означает возможность выполнения итерации — прохода — по содержимому таких структур данных с помощью цикла `for/of`, как было показано в подразделе 5.4.4:

```
let sum = 0;
for(let i of [1,2,3]) {     // Однократный проход по каждому значению
   sum += i;
}
sum                         // => 6
```

Итераторы также можно применять с операцией `...` для расширения или "распространения" итерируемого объекта в инициализаторе массива или в вызове функции, что демонстрировалось в подразделе 7.1.2:

```
let chars = [..."abcd"];    // chars == ["a", "b", "c", "d"]
let data = [1, 2, 3, 4, 5];
Math.max(...data)           // => 5
```

Итераторы можно использовать в деструктурирующем присваивании:

```
let purpleHaze = Uint8Array.of(255, 0, 255, 128);
let [r, g, b, a] = purpleHaze;          // a == 128
```

При выполнении итерации по объекту `Map` возвращаемыми значениями будут пары `[ключ, значение]`, которые хорошо сочетаются с деструктурирующим присваиванием в цикле `for/of`:

```
let m = new Map([["one", 1], ["two", 2]]);
for(let [k,v] of m) console.log(k, v);  // Выводится 'one 1' и 'two 2'
```

Если вы хотите организовать итерацию только по ключам или только по значениям, а не по их парам, тогда можете применять методы `keys()` и `values()`:

```
[...m]          // => [["one", 1], ["two", 2]]: итерация по умолчанию
[...m.entries()]  // => [["one", 1], ["two", 2]]: то же самое
                  // обеспечивает метод entries()
[...m.keys()]     // => ["one", "two"]: метод keys() выполняет
                  // итерацию только по ключам в Map
[...m.values()]   // => [1, 2]: метод values() выполняет итерацию
                  // только по значениям в Map
```

Наконец, ряд встроенных функций и конструкторов, которые обычно используются с объектами `Array`, в действительности реализованы (в ES6 и последующих версиях) для приема произвольных итераторов. Конструктор `Set()` — один из API-интерфейсов такого рода:

```
// Строки итерируемы, поэтому два множества одинаковы:
new Set("abc")     // => new Set(["a", "b", "c"])
```

В настоящей главе объясняется работа итераторов и методика создания собственных структур данных, которые являются итерируемыми. После обсуждения базовых итераторов в главе раскрываются генераторы — мощное новое средство ES6, которое главным образом применяется в качестве очень легкого способа создания итераторов.

12.1. Особенности работы итераторов

Цикл `for/of` и операция распространения без проблем работают с итерируемыми объектами, но немаловажно понимать, что на самом деле должно произойти для того, чтобы заставить итерацию работать. 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 *итератора*, 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`. Таким образом, простой цикл `for/of` по итерируемому объекту `iterable` может быть записан в сложной форме:

```
let iterable = [99];
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 вида, 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]
```

12.2. Реализация итерируемых объектов

Итерируемые объекты настолько полезны в ES6, что вы должны подумать о том, чтобы сделать собственные типы данных итерируемыми всякий раз, когда они представляют что-нибудь, допускающее итерацию. Классы `Range`, показанные в примерах 9.2 и 9.3 из главы 9, были итерируемыми. Классы `Range` применяли генераторные функции, чтобы превратить себя в итерируемые классы. Генераторы будут описаны далее в главе, но сначала мы реализуем класс `Range` еще раз, сделав его итерируемым без помощи генератора.

Чтобы сделать класс итерируемым, потребуется реализовать метод с символьным именем `Symbol.iterator`, который должен возвращать объект итератора, имеющий метод `next()`. А метод `next()` обязан возвращать объект результата итерации, который имеет свойство `value` и/или булевское свойство `done`. В примере 12.1 реализован итерируемый класс `Range` и показано, как создавать итерируемый объект, объект итератора и объект результата итерации.

Пример 12.1. Итерируемый числовой класс Range

```
/*
 * 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) {
        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]
```

В дополнение к превращению своих классов в итерируемые иногда весьма полезно определить функции, которые возвращают итерируемые значения. Рассмотрим такие итерируемые альтернативы методам `map()` и `filter()` массивов JavaScript:

```
// 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]
```

```
// 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 итератором, и итерируемым.
        [Symbol.iterator]() { return this; },
        next() {
            for(;;) {
                let v = iterator.next();
                if (v.done || predicate(v.value)) {
                    return v;
                }
            }
        }
    };
}
// Отфильтровать диапазон, чтобы остались только четные числа.
[...filter(new Range(1,10), x => x % 2 === 0)]  // => [2,4,6,8,10]
```

Одна из ключевых особенностей итерируемых объектов заключается в том, что они в своей основе ленивые: если для получения следующего значения требуется вычисление, то вычисление может быть отложено до момента, когда значение фактически понадобится. Пусть, например, у вас есть очень длинная строка текста, которую вы хотите разбить на отдельные слова. Вы могли бы просто воспользоваться методом `split()` вашей строки, но в таком случае до того, как удастся задействовать даже первое слово, должна быть полностью обработана вся строка. И в итоге вы выделили бы много памяти для возвращенного массива и всех строк внутри него. Ниже приведен код функции, которая позволяет ленивым образом проходить по словам в строке, не сохраняя сразу их всех в памяти (в ES2020 эту функцию можно было бы реализовать гораздо легче с применением метода `matchAll()`, возвращающего итератор, который был описан в подразделе 11.3.2):

```
function words(s) {
    var r = /\s+|$/g;    // Соответствует одному или большему
                         // количеству пробелов или концу.
    r.lastIndex = s.match(/[^ ]/).index; // Начать сопоставление
                // с первого символа, отличающегося от пробела.
    return {    // Возвратить итерируемый объект итератора.
        [Symbol.iterator]() {         // Это делает итерируемый объект.
            return this;
        },
        next() {                      // Это делает объект итератора.
            let start = r.lastIndex;  // Продолжить там, где закончи-
                                      // лось последнее совпадение.
            if (start < s.length) {   // Если работа не закончена.
                let match = r.exec(s); // Соответствует следующей
                                      // границе слова.
```

```
            if (match) {   // Если она найдена, то возвратить слово.
                return { value: s.substring(start, match.index) };
            }
        }
        return { done: true };     // В противном случае указать,
                                   // что работа закончена.
    }
  };
}
[...words(" abc def ghi! ")]       // => ["abc", "def", "ghi!"]
```

12.2.1. "Закрытие" итератора: метод `return()`

Представим себе вариант итератора `words()` на JavaScript стороны сервера, который в своем аргументе принимает не исходную строку, а имя файла, открывает этот файл, читает из него строки и проходит по словам в этих строках. В большинстве операционных систем программы, которые открывают файлы для чтения из них, должны не забывать о закрытии файлов, когда чтение завершено, так что наш гипотетический итератор обязан гарантировать закрытие файла после того, как метод `next()` возвратит последнее слово в нем.

Но итераторы не всегда работают до конца: цикл `for/of` может быть прекращен 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.

Даже когда наш гипотетический итератор по словам в файле никогда не дорабатывает до конца, то ему все равно необходимо закрыть файл, который он открыл. По этой причине объекты итераторов могут реализовывать вместе с методом `next()` метод `return()`. Если итерация останавливается до того, как `next()` возвратил результат итерации со свойством `done`, установленным в `true` (чаще всего из-за преждевременного покидания цикла `for/of` через оператор `break`), тогда интерпретатор проверит, есть ли у объекта итератора метод `return()`. Если метод `return()` существует, то интерпретатор вызовет его без аргументов, давая итератору шанс закрыть файлы, освободить память и как-то иначе произвести очистку после себя. Метод `return()` обязан возвращать объект результата итерации. Свойства этого объекта игнорируются, но возвращение значения, отличающегося от объекта, приведет к ошибке.

Цикл `for/of` и операция распространения — действительно полезные средства JavaScript, так что при создании API-интерфейсов рекомендуется по возможности применять их. Но необходимость работать с итерируемым объектом, объектом его итератора и объектами результатов итераций несколько усложняет процесс. К счастью, как будет показано в остальных разделах главы, генераторы способны значительно упростить создание специальных итераторов.

12.3. 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()` заставляет 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 выполнению 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()]           // => [2,3,5,7]
let sum = 0;
for(let prime of oneDigitPrimes()) sum += prime;
sum                             // => 17
```

В показанном выше примере для определения генератора мы использовали оператор `function*`. Однако подобно обыкновенным функциям мы можем определять генераторы также в форме выражений. Нужно лишь поместить звездочку после ключевого слова `function`:

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

В классах и объектных литералах можно применять сокращенную запись, полностью опуская ключевое слово `function` при определении методов. Чтобы определить генератор в таком контексте, мы просто используем звездочку перед именем метода, где находилось бы ключевое слово `function`, если бы оно было указано:

```
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()]  // => ["x", "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, намного более короткой генераторной функцией `*[Symbol.iterator&rbrack;()`, которая выглядит следующим образом:

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

Чтобы увидеть такую основанную на генераторе функцию итератора в контексте, просмотрите пример 9.3 в главе 9.

12.3.1. Примеры генераторов

Генераторы более интересны, если они действительно *генерируют* выдаваемые значения, выполняя вычисление какого-нибудь вида. Скажем, ниже приведена генераторная функция, которая выдает числа Фибоначчи:

```
function* fibonacciSequence() {
    let x = 0, y = 1;
    for(;;) {
        yield y;
        [x, y] = [y, x+y];  // Примечание: деструктурирующее присваивание.
    }
}
```

Обратите внимание, что генераторная функция `fibonacciSequence()` содержит бесконечный цикл и постоянно выдает значения без возврата. В случае использования этого генератора с операцией распространения `...` цикл будет выполняться до тех пор, пока не исчерпается память и произойдет аварийный отказ программы. Тем не менее, соблюдая осторожность, данный генератор можно применять в цикле `for/of`:

```
// Возвращает n-тое число Фибоначчи.
function fibonacci(n) {
   for(let f of fibonacciSequence()) {
      if (n-- <= 0) return f;
   }
}
fibonacci(20)              // => 10946
```

Бесконечный генератор подобного рода становится более полезным, когда используется в генераторе `take()` следующего вида:

```
// Выдает первые n элементов указанного итерируемого объекта.
function* take(n, iterable) {
  let it = iterable[Symbol.iterator](); // Получить итератор для
                                        // итерируемого объекта.
  while(n-- > 0) {          // Цикл n 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, 5]
```

Вот еще одна полезная генераторная функция, которая чередует элементы множества итерируемых объектов:

```
// Для заданного массива итерируемых объектов выдает
// их элементы в чередующемся порядке.
function* zip(...iterables) {
   // Получить итератор для каждого итерируемого объекта.
   let iterators = iterables.map(i => i[Symbol.iterator]());
   let index = 0;
   while(iterators.length > 0) {      // Пока есть какие-то итераторы.
      if (index >= iterators.length) {// Если мы достигли последнего
                                      // итератора, тогда вернуться
         index = 0;                   // к первому итератору.
      }
      let item = iterators[index].next(); //Получить следующий элемент
                                 // от следующего итератора.
      if (item.done) {          // Если этот итератор закончил работу,
         iterators.splice(index, 1); // тогда удалить его из массива.
      }
      else {                     // Иначе
```

```
            yield item.value;     // выдать текущее значение
            index++;              // и перейти к следующему итератору.
        }
    }
}
// Чередовать элементы трех итерируемых объектов.
[...zip(oneDigitPrimes(),"ab",[0])]        // => [2,"a",0,3,"b",5,7]
```

12.3.2. `yield*` и рекурсивные генераторы

Помимо генератора `zip()`, определенного в предыдущем примере, полезно иметь аналогичную генераторную функцию, которая выдает элементы множества итерируемых объектов последовательно, не чередуя их. Мы могли бы написать такой генератор следующим образом:

```
function* sequence(...iterables) {
    for(let iterable of iterables) {
        for(let item of iterable) {
            yield item;
        }
    }
}
[...sequence("abc",oneDigitPrimes())]  // => ["a","b","c",2,3,5,7]
```

Процесс выдачи элементов какого-то другого итерируемого объекта достаточно распространен в генераторных функциях, поэтому в ES6 для него предусмотрен специальный синтаксис. Ключевое слово `yield*` похоже на `yield`, но вместо выдачи одиночного значения оно проходит по итерируемому объекту и выдает каждое результирующее значение. Генераторную функцию `sequence()`, которую мы применяли ранее, за счет использования `yield*` можно упростить:

```
function* sequence(...iterables) {
    for(let iterable of iterables) {
        yield* iterable;
    }
}
[...sequence("abc",oneDigitPrimes())]  // => ["a","b","c",2,3,5,7]
```

Метод `forEach()` массива часто оказывается элегантным способом прохода в цикле по элементам массива, поэтому у вас может возникнуть соблазн реализовать функцию `sequence()`, как показано ниже:

```
function* sequence(...iterables) {
    iterables.forEach(iterable => yield* iterable );  // Ошибка!
}
```

Однако такой код работать не будет. Ключевые слова `yield` и `yield*` могут применяться только внутри генераторных функций, но вложенная стрелочная функция в данном коде является обыкновенной, а не генераторной функцией `function*`, так что использование `yield` в ней не разрешено.

Ключевое слово `yield*` можно применять с итерируемым объектом любого вида, включая итерируемые объекты, которые реализованы с помощью генераторов. Таким образом, ключевое слово `yield*` позволяет определять рекурсивные генераторы, и вы можете использовать это средство, например, чтобы обеспечить простой нерекурсивный обход рекурсивно определенной древовидной структуры.

12.4. Расширенные возможности генераторов

Генераторные функции чаще всего применяются для создания итераторов, но фундаментальная особенность генераторов заключается в том, что они позволяют приостанавливать вычисление, выдавать промежуточные результаты и позже возобновлять вычисление. Это означает, что генераторы обладают возможностями, выходящими за рамки итераторов, и мы исследуем их в последующих разделах.

12.4.1. Возвращаемое значение генераторной функции

Генераторные функции, встречавшиеся до сих пор, не имели операторов `return`, но даже если имели, то они использовались для преждевременного возврата, а не для того, что возвратить значение. Тем не менее, как и любая функция, генераторная функция способна возвращать значение. Для понимания того, что происходит в этом случае, вспомните, каким образом работает итерация. Возвращаемым значением функции `next()` является объект, который имеет свойство `value` и/или свойство `done`. В типичных итераторах и генераторах, если определено свойство `value`, то свойство `done` не определено или равно `false`. Если свойство `done` равно `true`, тогда свойство `value` не определено. Но в ситуации с генератором, который возвращает значение, финальный вызов `next` возвращает объект, в котором определены оба свойства, `value` и `done`. Свойство `value` хранит возвращаемое значение генераторной функции, а свойство `done` равно `true`, указывая на то, что больше нет значения для итерации. Цикл `for/of` и операция распространения игнорируют это финальное значение, но оно доступно коду, в котором производится итерация вручную с помощью явных вызовов `next()`:

```
function *oneAndDone() {
   yield 1;
   return "done";
}
// При нормальной итерации возвращаемое значение не появляется.
[...oneAndDone()]          // => [1]
// Но оно доступно в случае явного вызова next().
let generator = oneAndDone();
generator.next()           // => { value: 1, done: false}
generator.next()           // => { value: "done", done: true }
// Если генератор уже закончил работу, то возвращаемое значение
// больше не возвращается.
generator.next()           // => { value: undefined, done: true }
```

12.4.2. Значение выражения `yield`

В предыдущем обсуждении мы обходились с `yield` как с оператором, который принимает значение, но не имеет собственного значения. Однако на самом деле `yield` является выражением и может иметь значение.

Когда вызывается метод `next()` генератора, генераторная функция выполняется до тех пор, пока не достигает выражения `yield`. Выражение, указанное после ключевого слова `yield`, вычисляется и его значение становится возвращаемым значением вызова `next()`. В данной точке генераторная функция останавливает выполнение прямо на середине вычисления выражения `yield`. Когда метод `next()` генератора вызывается в следующий раз, переданный `next()` аргумент становится значением выражения `yield`, вычисление которого было приостановлено. Таким образом, генератор возвращает значения вызывающему коду с помощью `yield`, а вызывающий код передает значения генератору посредством `next()`. Генератор и вызывающий код — два отдельных потока выполнения, передающие значения (и управление) туда и обратно. Следующий код иллюстрирует сказанное:

```
function* smallNumbers() {
   console.log("next() вызывается первый раз; аргумент отбрасывается");
   let y1 = yield 1;    // y1 == "b"
   console.log("next() вызывается второй раз с аргументом", y1);
   let y2 = yield 2;    // y2 == "c"
   console.log("next() вызывается третий раз с аргументом", y2);
   let y3 = yield 3;    // y3 == "d"
   console.log("next() вызывается четвертый раз с аргументом", y3);
   return 4;
}
let g = smallNumbers();
console.log("генератор создан; никакой код пока не выполнялся");
let n1 = g.next("a");   // n1.value == 1
console.log("генератор выдает", n1.value);
let n2 = g.next("b");   // n2.value == 2
console.log("генератор выдает", n2.value);
let n3 = g.next("c");   // n3.value == 3
console.log("генератор выдает", n3.value);
let n4 = g.next("d");   // n4 == { value: 4, done: true }
console.log("генератор возвращает", n4.value);
```

Во время выполнения код производит показанный ниже вывод, который демонстрирует обмен значениями между двумя блоками кода:

```
генератор создан; никакой код пока не выполнялся
next() вызывается первый раз; аргумент отбрасывается
генератор выдает 1
next() вызывается второй раз с аргументом b
генератор выдает 2
next() вызывается третий раз с аргументом c
генератор выдает 3
next() вызывается четвертый раз с аргументом d
генератор возвращает 4
```

Обратите внимание на асимметрию в этом коде. Первый вызов `next()` запускает генератор, но значение, переданное данному вызову, оказывается недоступным генератору.

12.4.3. Методы `return()` и `throw()` генератора

Вы уже видели, что можно получать значения, выдаваемые или возвращаемые генераторной функцией. Кроме того, можно передавать значения выполняющемуся генератору, указывая их при вызове метода `next()` генератора.

Помимо предоставления входных данных генератору с помощью `next()` можно также изменять поток управления внутри генератора, вызывая его методы `return()` и `throw()`. Как должно быть понятно по их именам, вызов упомянутых методов генератора заставляет его возвратить значение или сгенерировать исключение, как если бы следующим оператором был `return` или `throw`.

Как объяснялось ранее в главе, если итератор определяет метод `return()` и итерация останавливается преждевременно, тогда интерпретатор автоматически вызывает метод `return()`, чтобы дать итератору возможность закрыть файлы или выполнить другую очистку. В случае генераторов нельзя определить специальный метод `return()` для поддержки очистки, но можно структурировать код генератора так, чтобы задействовать оператор `try/finally`, который гарантирует выполнение необходимой очистки (в блоке `finally`), когда происходит возврат из генератора. За счет принудительного возврата из генератора его встроенный метод `return()` обеспечивает выполнение кода очистки, когда генератор больше не будет использоваться.

Подобно тому, как метод `next()` генератора позволяет передавать произвольные значения выполняющемуся генератору, метод `throw()` генератора предоставляет нам способ отправки генератору произвольных сигналов (в форме исключений). Вызов метода `throw()` всегда инициирует исключение внутри генератора. Но если генераторная функция написана с соответствующим кодом обработки исключений, то исключение не обязано быть фатальным, а взамен может служить средством изменения поведения генератора. Представьте себе, скажем, генератор счетчика, который выдает строго возрастающую последовательность целых чисел. Его можно было бы реализовать так, чтобы исключение, отправленное с помощью `throw()`, сбрасывало счетчик в ноль.

Когда генератор применяет `yield*` для выдачи значений из какого-то другого итерируемого объекта, то вызов метода `next()` генератора приводит к вызову метода `next()` итерируемого объекта. То же самое справедливо в отношении методов `return()` и `throw()`. Если генератор использует `yield*` с итерируемым объектом, в котором эти методы определены, тогда вызов метода `return()` или `throw()` генератора приводит к вызову метода `return()` или `throw()` итератора по очереди. Все итераторы *обязаны* иметь метод `next()`. Итераторы, которые нуждаются в очистке после незавершенной итерации, *должны* определять метод `return()`. И любой итератор *может* определять метод `throw()`, хотя я не знаю никаких практических причин для этого.

12.4.4. Финальное замечание о генераторах

Генераторы — очень мощная обобщенная структура управления. Они дают нам возможность приостанавливать вычисление с помощью `yield` и снова перезапускать 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.

Попытки делать такие вещи посредством генераторов приводят к появлению кода, который невероятно сложно понять или объяснить. Тем не менее, это делалось, и единственным действительно практичным сценарием применения было управление асинхронным кодом. Однако теперь в JavaScript есть ключевые слова `async` и `await` (см. главу 13), предназначенные именно для такой цели, и больше нет никаких причин злоупотреблять генераторами подобным образом.

12.5. Резюме

Ниже перечислены основные моменты, которые рассматривались в главе.

- Цикл `for/of` и операция распространения `...` работают с итерируемыми объектами.
- Объект является итерируемым, если он имеет метод с символьным именем `[Symbol.iterator]`, который возвращает объект итератора.
- Объект итератора имеет метод `next()`, который возвращает объект результата итерации.
- Объект результата итерации имеет свойство `value`, которое хранит следующее проходимое значение при его наличии. Если итерация закончилась, тогда объект результата должен иметь свойство `done`, установленное в `true`.
- Вы можете реализовывать собственные итерируемые объекты, определяя метод `[Symbol.iterator]()`, который возвращает объект с методом `next()`, возвращающим объекты результатов итерации. Вы также можете реализовывать функции, которые принимают аргументы итераторов и возвращают значения итераторов.
- Генераторные функции (функции, определенные с помощью `function*` вместо `function`) представляют собой еще один способ определения итераторов.
- Когда вы вызываете генераторную функцию, ее тело выполняется не сразу; взамен возвращается значение, которое является объектом итератора. Каждый раз, когда вызывается метод `next()` этого итератора, выполняется еще одна порция генераторной функции.
- В генераторных функциях можно использовать ключевое слово `yield` для указания значений, которые возвращаются итератором. Каждый вызов `next()` приводит к тому, что генераторная функция выполняется до следующего выражения `yield`. Затем значение выражения `yield` становится значением, возвращаемым итератором. Когда выражений `yield` больше нет, тогда происходит возврат из генераторной функции и итерация завершается.