Перебираемые объекты

Перебираемые (или *итерируемые*) объекты – это концепция, которая позволяет использовать любой объект в цикле `for..of`.

Конечно же, сами массивы являются перебираемыми объектами. Но есть и много других встроенных перебираемых объектов, например, строки.

Если объект не является массивом, но представляет собой коллекцию каких-то элементов, то удобно использовать цикл `for..of` для их перебора, так что давайте посмотрим, как это сделать.

Symbol.iterator

Мы легко поймём принцип устройства перебираемых объектов, создав один из них.

Например, у нас есть объект. Это не массив, но он выглядит подходящим для `for..of`.

Например, объект `range`, который представляет собой диапазон чисел:

```
let range = {
  from: 1,
  to: 5
};

// Мы хотим, чтобы работал for..of:
// for(let num of range) ... num=1,2,3,4,5
```

Чтобы сделать `range` итерируемым (и позволить `for..of` работать с ним), нам нужно добавить в объект метод с именем `Symbol.iterator` (специальный встроенный `Symbol`, созданный как раз для этого).

1. Когда цикл `for..of` запускается, он вызывает этот метод один раз (или выдаёт ошибку, если метод не найден). Этот метод должен вернуть *итератор* – объект с методом `next`.
2. Дальше `for..of` работает *только с этим возвращённым объектом*.
3. Когда `for..of` хочет получить следующее значение, он вызывает метод `next()` этого объекта.
4. Результат вызова `next()` должен иметь вид `{done: Boolean, value: any}`, где `done=true` означает, что итерация закончена, в противном случае `value` содержит очередное значение.

Вот полная реализация `range` с пояснениями:

```
let range = {
  from: 1,
  to: 5
};

// 1. вызов for..of сначала вызывает эту функцию
range[Symbol.iterator] = function() {

  // ...она возвращает объект итератора:
  // 2. Далее, for..of работает только с этим итератором
  return {
    current: this.from,
    last: this.to,

    // 3. next() вызывается на каждой итерации цикла fo
    next() {
```


```
17        // 4. он должен вернуть значение в виде объекта {
18        if (this.current <= this.last) {
19          return { done: false, value: this.current++ };
20        } else {
21          return { done: true };
22        }
23      }
24    };
25  };
26
27  // теперь работает!
28  for (let num of range) {
29    alert(num); // 1, затем 2, 3, 4, 5
30  }
```

Обратите внимание на ключевую особенность итераторов: разделение ответственности.

- У самого `range` нет метода `next()`.
- Вместо этого другой объект, так называемый «итератор», создаётся вызовом `range[Symbol.iterator]()`, и именно его `next()` генерирует значения.

Таким образом, итератор отделён от самого итерируемого объекта.

Технически мы можем объединить их и использовать сам `range` как итератор, чтобы упростить код.

Например, вот так:

```
1  let range = {
2    from: 1,
3    to: 5,
4
5    [Symbol.iterator]() {
6      this.current = this.from;
7      return this;
8    },
9
10   next() {
11     if (this.current <= this.to) {
12       return { done: false, value: this.current++ };
13     } else {
14       return { done: true };
15     }
16   }
17 };
18
19 for (let num of range) {
20   alert(num); // 1, затем 2, 3, 4, 5
21 }
```

Теперь `range[Symbol.iterator]()` возвращает сам объект `range`: у него есть необходимый метод `next()`, и он запоминает текущее состояние итерации в `this.current`. Короче? Да. И иногда такой способ тоже хорош.

Недостаток такого подхода в том, что теперь мы не можем использовать этот объект в двух параллельных циклах `for..of`: у них будет общее текущее состояние итерации, потому что теперь существует лишь один итератор – сам объект. Но необходимость в двух циклах `for..of`, выполняемых одновременно, возникает редко, даже при наличии асинхронных операций.

Бесконечные итераторы

Можно сделать бесконечный итератор. Например, `range` будет бесконечным при `range.to = Infinity`. Или мы можем создать итерируемый объект, который генерирует бесконечную последовательность псевдослучайных чисел. Это бывает полезно.

Метод `next` не имеет ограничений, он может возвращать всё новые и новые значения, это нормально.

Конечно же, цикл `for..of` с таким итерируемым объектом будет бесконечным. Но мы всегда можем прервать его, используя `break`.

Строка – перебираемый объект

Среди встроенных перебираемых объектов наиболее широко используются массивы и строки.

Для строки `for..of` перебирает символы:

```
for (let char of "test") {
  // срабатывает 4 раза: по одному для каждого символа
  alert( char ); // t, затем e, затем s, затем t
}
```

И он работает корректно даже с суррогатными парами!

```
let str = '𝒳😂';
for (let char of str) {
    alert( char ); // 𝒳, а затем 😂
}
```

Явный вызов итератора

Чтобы понять устройство итераторов чуть глубже, давайте посмотрим, как их использовать явно.

Мы будем перебирать строку точно так же, как цикл `for..of`, но вручную, прямыми вызовами. Нижеприведённый код получает строковый итератор и берёт из него значения:

```
let str = "Hello";

// делает то же самое, что и
// for (let char of str) alert(char);

let iterator = str[Symbol.iterator]();

while (true) {
  let result = iterator.next();
  if (result.done) break;
  alert(result.value); // выводит символы один за други
}
```

Такое редко бывает необходимо, но это даёт нам больше контроля над процессом, чем `for..of`. Например, мы можем разбить процесс итерации на части: перебрать немного элементов, затем остановиться, сделать что-то ещё и потом продолжить.

Итерируемые объекты и псевдомассивы

Есть два официальных термина, которые очень похожи, но в то же время сильно различаются. Поэтому убедитесь, что вы как следует поняли их, чтобы избежать путаницы.

- *Итерируемые объекты* – это объекты, которые реализуют метод `Symbol.iterator`, как было описано выше.
- *Псевдомассивы* – это объекты, у которых есть индексы и свойство `length`, то есть, они выглядят как массивы.

При использовании JavaScript в браузере или других окружениях мы можем встретить объекты, которые являются итерируемыми или псевдомассивами, или и тем, и другим.

Например, строки итерируемы (для них работает `for..of`) и являются псевдомассивами (они индексированы и есть `length`).

Но итерируемый объект может не быть псевдомассивом. И наоборот: псевдомассив может не быть итерируемым.

Например, объект `range` из примера выше – итерируемый, но не является псевдомассивом, потому что у него нет индексированных свойств и `length`.

А вот объект, который является псевдомассивом, но его нельзя итерировать:

```
let arrayLike = { // есть индексы и свойство length
  0: "Hello",
  1: "World",
  length: 2
};

// Ошибка (отсутствует Symbol.iterator)
for (let item of arrayLike) {}
```

Что у них общего? И итерируемые объекты, и псевдомассивы – это обычно *не массивы*, у них нет методов `push`, `pop` и т.д. Довольно неудобно, если у нас есть такой объект и мы хотим работать с ним как с массивом. Например, мы хотели бы работать с `range`, используя методы массивов. Как этого достичь?

Array.from

Есть универсальный метод Array.from, который принимает итерируемый объект или псевдомассив и делает из него «настоящий» `Array`. После этого мы уже можем использовать методы массивов.

Например:

```
let arrayLike = {
  0: "Hello",
  1: "World",
  length: 2
};

let arr = Array.from(arrayLike); // (*)
alert(arr.pop()); // World (метод работает)
```

`Array.from` в строке `(*)` принимает объект, проверяет, является ли он итерируемым объектом или псевдомассивом, затем создаёт новый массив и копирует туда все элементы.

То же самое происходит с итерируемым объектом:

```
// range взят из примера выше
let arr = Array.from(range);
alert(arr); // 1,2,3,4,5 (преобразование массива через
```

Полный синтаксис `Array.from` позволяет указать необязательную «трансформирующую» функцию:

```
Array.from(obj[, mapFn, thisArg])
```

Необязательный второй аргумент может быть функцией, которая будет применена к каждому элементу перед добавлением в массив, а `thisArg` позволяет установить `this` для этой функции.

Например:

```
// range взят из примера выше

// возводим каждое число в квадрат
let arr = Array.from(range, num => num * num);

alert(arr); // 1,4,9,16,25
```

Здесь мы используем `Array.from`, чтобы превратить строку в массив её элементов:

```
let str = '𝒳😂';

// разбивает строку на массив её элементов
let chars = Array.from(str);

alert(chars[0]); // 𝒳
alert(chars[1]); // 😂
alert(chars.length); // 2
```

В отличие от `str.split`, этот метод в работе опирается на итерируемость строки, и поэтому, как и `for..of`, он корректно работает с суррогатными парами.

Технически это то же самое, что и:

```
let str = '𝒳😂';

let chars = []; // Array.from внутри себя выполняет тот
for (let char of str) {
  chars.push(char);
}

alert(chars);
```

...Но гораздо короче.

Мы можем даже создать `slice`, который поддерживает суррогатные пары:

```
function slice(str, start, end) {
  return Array.from(str).slice(start, end).join('');
}

let str = '𝒳😂𩷶';

alert( slice(str, 1, 3) ); // 😂𩷶

// а вот встроенный метод не поддерживает суррогатные п
alert( str.slice(1, 3) ); // мусор (две части различных
```

Итого

Объекты, которые можно использовать в цикле `for..of`, называются *итерируемыми*.

- Технически итерируемые объекты должны иметь метод `Symbol.iterator`.
  - Результат вызова `obj[Symbol.iterator]` называется *итератором*. Он управляет процессом итерации.

  - Итератор должен иметь метод `next()`, который возвращает объект `{done: Boolean, value: any}`, где `done:true` сигнализирует об окончании процесса итерации, в противном случае `value` – следующее значение.
- Метод `Symbol.iterator` автоматически вызывается циклом `for..of`, но можно вызвать его и напрямую.
- Встроенные итерируемые объекты, такие как строки или массивы, также реализуют метод `Symbol.iterator`.
- Строковый итератор знает про суррогатные пары.

Объекты, имеющие индексированные свойства и `length`, называются *псевдомассивами*. Они также могут иметь другие свойства и методы, но у них нет встроенных методов массивов.

Если мы заглянем в спецификацию, мы увидим, что большинство встроенных методов рассчитывают на то, что они будут работать с итерируемыми объектами или псевдомассивами вместо «настоящих» массивов, потому что эти объекты более абстрактны.

`Array.from(obj[, mapFn, thisArg])` создаёт настоящий `Array` из итерируемого объекта или псевдомассива `obj`, и затем мы можем применять к нему методы массивов. Необязательные аргументы `mapFn` и `thisArg` позволяют применять функцию с задаваемым контекстом к каждому элементу.

Комментарии

перед тем как писать...