

# JavaScript - генераторы

Symbol

Итераторы

Генераторы

Асинхронные итераторы и генераторы

# Тип данных Symbol

Тип данных `Symbol` в JavaScript представляет собой уникальный и неизменяемый примитивный тип данных, который был добавлен в стандарт ECMAScript 2015 (ES6). Символы часто используются для создания уникальных идентификаторов для свойств объектов, что предотвращает конфликт имен свойств.

Основные особенности `Symbol`:

- Уникальность
- Описание (описание символа)
- Использование в объектах
- Итерация и перечисление
- Глобальные символы

# Уникальность

Каждый символ ункален и может быть создан с помощью функции `Symbol()`.

Два символа, созданные с одинаковым описанием, не равны друг другу.

```
let sym1 = Symbol('description');
```

```
let sym2 = Symbol('description');
```

```
console.log(sym1 === sym2); // false
```

# Описание (описание символа)

При создании символа можно указать описание (или метку), которое помогает при отладке, но не влияет на уникальность символа.

```
let sym = Symbol('mySymbol');  
  
console.log(sym.description); // 'mySymbol'
```

# Использование в объектах

Символы часто используются как уникальные ключи для свойств объектов.

```
let mySymbol = Symbol('uniqueKey');
```

```
let obj = {  
    [mySymbol]: 'value'  
};
```

```
console.log(obj[mySymbol]); // 'value'
```

# Итерация и перечисление

Свойства объектов, ключами которых являются символы, не будут отображаться при итерации по объекту с помощью `for...in`, `Object.keys()`, `Object.getOwnPropertyNames()`, но они будут видны с помощью `Object.getOwnPropertySymbols()`.

```
let sym = Symbol('hidden');
let obj = {
  [sym]: 'hiddenValue',
  visible: 'visibleValue'
};

for (let key in obj) {
  console.log(key); // 'visible'
}

console.log(Object.keys(obj)); // ['visible']
console.log(Object.getOwnPropertySymbols(obj)); // [ Symbol(hidden) ]
```

# Глобальные символы

Существуют глобальные символы, которые могут быть доступны в любой части программы с использованием `Symbol.for()` и `Symbol.keyFor()`.

```
let globalSym1 = Symbol.for('shared');
```

```
let globalSym2 = Symbol.for('shared');
```

```
console.log(globalSym1 === globalSym2); // true
```

```
console.log(Symbol.keyFor(globalSym1)); // 'shared'
```

# Встроенные символы (Well-Known Symbols)

JavaScript также определяет ряд встроенных символов, которые позволяют изменять поведение некоторых встроенных методов и операций. Например:

- `Symbol.iterator` для создания итераторов для объектов.
- `Symbol.toStringTag` для изменения результата `Object.prototype.toString`.

```
class Collection {
    constructor() {
        this.items = [];
    }

    add(item) {
        this.items.push(item);
    }

    [Symbol.iterator]() {
        let index = 0;
        let items = this.items;

        return {
            next: function() {
                if (index < items.length) {
                    return { value: items[index++], done: false };
                } else {
                    return { done: true };
                }
            }
        };
    }
}

let collection = new Collection();
collection.add('item1');
collection.add('item2');

for (let item of collection) {
    console.log(item); // 'item1', 'item2'
}
```

# Итераторы

Итераторы в JavaScript — это объекты, которые позволяют проходить через коллекцию элементов по одному за раз. Итераторы предоставляют метод `next()`, который возвращает следующий элемент последовательности в виде объекта с двумя свойствами: `value` (следующее значение) и `done` (логическое значение, указывающее, закончена ли итерация).

# Создание итератора

Пример с обычным объектом

Вот пример создания простого итератора для массива

```
function createIterator(array) {
  let index = 0;
  return {
    next: function() {
      if (index < array.length) {
        return { value: array[index++], done: false };
      } else {
        return { value: undefined, done: true };
      }
    }
  }
}

const myArray = ['a', 'b', 'c'];
const iterator = createIterator(myArray);

console.log(iterator.next()); // { value: 'a', done: false }
console.log(iterator.next()); // { value: 'b', done: false }
console.log(iterator.next()); // { value: 'c', done: false }
console.log(iterator.next()); // { value: undefined, done: true }
```

# Символ `Symbol.iterator`

`Symbol.iterator` — это встроенный символ в JavaScript, который используется для определения итераторов для объектов. Итератор — это объект, который определяет метод `next()`, возвращающий следующий элемент последовательности.

Когда объект имеет метод с ключом `Symbol.iterator`, он становится итерируемым, и его можно использовать в конструкциях, которые работают с итерациями, таких как `for...of` циклы, оператор `spread`, деструктуризация и другие.

# Символ Symbol.iterator

Пример создания итерируемого объекта

Вот пример объекта, который можно итерировать с использованием Symbol.iterator

```
const myIterable = {
  items: ['item1', 'item2', 'item3'],
  [Symbol.iterator]: function() {
    let index = 0;
    let items = this.items;
    return {
      next: function() {
        if (index < items.length) {
          return { value: items[index++], done: false };
        } else {
          return { value: undefined, done: true };
        }
      }
    };
  }
};

for (const item of myIterable) {
  console.log(item); // 'item1', 'item2', 'item3'
}
```

# Использование с другими итераторными конструкциями

Оператор spread:

```
const arr = [...myIterableObject];
console.log(arr); // [1, 2, 3, 4, 5];
```

Деструктуризация:

```
const [first, second, ...rest] = myIterableObject;
console.log(first); // 1
console.log(second); // 2
console.log(rest); // [3, 4, 5]
```

Array.from:

```
const arrayFromIterable = Array.from(myIterableObject);
console.log(arrayFromIterable); // [1, 2, 3, 4, 5]
```

# Итерируемые объекты

Итерируемые объекты (iterable objects) — это объекты, которые реализуют метод [Symbol.iterator]. Этот метод должен возвращать объект-итератор с методом next(), который возвращает следующий элемент последовательности в виде объекта с двумя свойствами: value (следующее значение) и done (логическое значение, указывающее, закончена ли итерация).

# Встроенные итераторы в JavaScript

Многие встроенные объекты в JavaScript уже имеют реализованные итераторы через `Symbol.iterator`:

- Массивы
- Строки
- Map и Set

```
const array = [10, 20, 30];
const iterator = array[Symbol.iterator]();

console.log(iterator.next()); // { value: 10, done: false }
console.log(iterator.next()); // { value: 20, done: false }
console.log(iterator.next()); // { value: 30, done: false }
console.log(iterator.next()); // { value: undefined, done: true }
```

```
const set = new Set([1, 2, 3]);
const setIterator = set[Symbol.iterator]();

console.log(setIterator.next()); // { value: 1, done: false }
console.log(setIterator.next()); // { value: 2, done: false }
console.log(setIterator.next()); // { value: 3, done: false }
console.log(setIterator.next()); // { value: undefined, done: true }

const map = new Map([[['a', 1], ['b', 2]]]);
const mapIterator = map[Symbol.iterator]();

console.log(mapIterator.next()); // { value: ['a', 1], done: false }
console.log(mapIterator.next()); // { value: ['b', 2], done: false }
console.log(mapIterator.next()); // { value: undefined, done: true }

const str = 'hello';
const iterator = str[Symbol.iterator]();

console.log(iterator.next()); // { value: 'h', done: false }
console.log(iterator.next()); // { value: 'e', done: false }
console.log(iterator.next()); // { value: 'l', done: false }
console.log(iterator.next()); // { value: 'l', done: false }
console.log(iterator.next()); // { value: 'o', done: true }
```

# Псевдомассивы

Псевдомассивы (array-like objects) — это объекты, которые имеют числовые индексы и свойство `length`, но не обладают методами массивов, такими как `push()`, `pop()`, `forEach()` и т.д. Чаще всего они появляются в виде объектов, возвращаемых методами, работающими с DOM.

# Примеры псевдомассивов:

**arguments** объект (внутри функции):

```
function example() {  
    console.log(arguments); // псевдомассив  
    for (let i = 0; i < arguments.length; i++) {  
        console.log(arguments[i]);  
    }  
}  
example('a', 'b', 'c'); // 'a', 'b', 'c'
```

**NodeList** объект (например, результат `document.querySelectorAll`):

```
const nodeList = document.querySelectorAll('div');  
console.log(nodeList); // псевдомассив  
for (let i = 0; i < nodeList.length; i++) {  
    console.log(nodeList[i]); // каждый элемент 'div'  
}
```

# Преобразование псевдомассивов в массивы

Псевдомассивы можно преобразовать в настоящие массивы, чтобы использовать все методы массивов.

Использование `Array.from`

Использование оператора `spread`

```
function example() {  
  const argsArray = Array.from(arguments);  
  console.log(argsArray); // ['a', 'b', 'c']  
}  
example('a', 'b', 'c');
```

```
function example() {  
  const argsArray = [...arguments];  
  console.log(argsArray); // ['a', 'b', 'c']  
}  
example('a', 'b', 'c');
```

# Различия между итерируемыми объектами и псевдомассивами

## Итерируемые объекты:

- Имеют метод [Symbol.iterator].
- Можно использовать в конструкциях, работающих с итераторами, таких как for...of, spread и т.д.

## Псевдомассивы:

- Имеют числовые индексы и свойство length.
- Не обладают методами массивов.
- Чаще всего требуют преобразования в массив для использования методов массивов.

# Генераторы

Генераторы в JavaScript — это особый тип функции, который позволяет приостанавливать и возобновлять выполнение. Генераторы создаются с помощью ключевого слова `function*` и могут использовать оператор `yield` для выдачи значений. Генераторы возвращают итератор, который можно использовать для последовательного получения значений.

# Основные концепции генераторов

**Объявление генератора:** Используется `function*` для объявления генератора.

**Использование генератора:** Генератор возвращает объект-итератор с методом `next()`.

**Оператор `yield`:** Оператор `yield` используется для приостановки выполнения генератора и возврата значения.

**Возобновление выполнения:** Выполнение генератора возобновляется с места, где было приостановлено оператором `yield`.

```
function* myGenerator() {  
    yield 1;  
    yield 2;  
    yield 3;  
}
```

```
const gen = myGenerator();  
console.log(gen.next()); // { value: 1, done: false }  
console.log(gen.next()); // { value: 2, done: false }  
console.log(gen.next()); // { value: 3, done: false }  
console.log(gen.next()); // { value: undefined, done: true }
```

# Генераторы с циклом

```
function* countTo(n) {  
    for (let i = 1; i <= n; i++) {  
        yield i;  
    }  
}  
  
const counter = countTo(5);  
for (let num of counter) {  
    console.log(num); // 1, 2, 3, 4, 5  
}
```

# Использование генераторов для создания итераторов

Генераторы могут быть использованы  
для создания итераторов для  
пользовательских объектов

```
const myObject = {
  items: ['x', 'y', 'z'],
  *[Symbol.iterator]() {
    for (let item of this.items) {
      yield item;
    }
  }
};

for (const item of myObject) {
  console.log(item); // 'x', 'y', 'z'
}
```

# Асинхронные итераторы и генераторы

Асинхронные итераторы и генераторы в JavaScript позволяют работать с асинхронными последовательностями данных, упрощая управление асинхронными операциями, такими как сетевые запросы, чтение файлов и другие IO-операции. Асинхронные итераторы были добавлены в ECMAScript 2018 и предоставляют стандартный способ обхода асинхронных коллекций.

# Асинхронные итераторы

Асинхронные итераторы — это объекты, которые реализуют метод `[Symbol.asyncIterator]` и возвращают асинхронный итератор. Асинхронный итератор имеет метод `next()`, который возвращает промис, резолвящийся в объект с двумя свойствами: `value` и `done`.

```
const asyncIterable = {
  data: [1, 2, 3],
  [Symbol.asyncIterator]() {
    let index = 0;
    return {
      next: async () => {
        if (index < this.data.length) {
          await new Promise(resolve => setTimeout(resolve, 1000)); // имитация задержки
          return { value: this.data[index++], done: false };
        } else {
          return { value: undefined, done: true };
        }
      }
    };
  }
};

(async () => {
  for await (const item of asyncIterable) {
    console.log(item); // 1, 2, 3 (с задержкой в 1 секунду между элементами)
  }
})();
```

# Асинхронные генераторы

Асинхронные генераторы — это функции, которые позволяют использовать `await` внутри себя и возвращают асинхронный итератор. Асинхронные генераторы создаются с помощью `async function*`\*

```
async function* asyncGenerator() {
    yield await new Promise(resolve => setTimeout(() => resolve(1), 1000));
    yield await new Promise(resolve => setTimeout(() => resolve(2), 1000));
    yield await new Promise(resolve => setTimeout(() => resolve(3), 1000));
}

(async () => {
    for await (const value of asyncGenerator()) {
        console.log(value); // 1, 2, 3 (с задержкой в 1 секунду между элементами)
    }
})();
```

# Использование асинхронных генераторов

Асинхронные генераторы упрощают работу с последовательностями асинхронных операций. Они позволяют писать код, который легко читать и поддерживать.

```
async function* fetchUrls(urls) {
  for (const url of urls) {
    const response = await fetch(url);
    const data = await response.json();
    yield data;
  }
}

(async () => {
  const urls = [
    'https://jsonplaceholder.typicode.com/posts/1',
    'https://jsonplaceholder.typicode.com/posts/2',
    'https://jsonplaceholder.typicode.com/posts/3'
  ];

  for await (const data of fetchUrls(urls)) {
    console.log(data);
  }
})();
```

# Асинхронные итераторы и символы

Асинхронные итераторы используют символ `Symbol.asyncIterator` для обозначения метода, который возвращает асинхронный итератор. Это аналогично `Symbol.iterator` для синхронных итераторов.

```
const customAsyncIterable = {
  [Symbol.asyncIterator]: async function* () {
    yield await new Promise(resolve => setTimeout(() => resolve('A'), 1000));
    yield await new Promise(resolve => setTimeout(() => resolve('B'), 1000));
    yield await new Promise(resolve => setTimeout(() => resolve('C'), 1000));
  }
};

(async () => {
  for await (const value of customAsyncIterable) {
    console.log(value); // 'A', 'B', 'C' (с задержкой в 1 секунду между элементами)
  }
})();
```

# Обработка ошибок в асинхронных генераторах

```
async function* errorHandlingGenerator() {
  try {
    const result1 = await new Promise((resolve, reject) => setTimeout(() => reject(new Error('Error 1')), 1000));
    yield result1;
  } catch (error) {
    console.error(error); // Обработка ошибки 'Error 1'
  }

  try {
    const result2 = await new Promise((resolve, reject) => setTimeout(() => resolve('Result 2'), 1000));
    yield result2;
  } catch (error) {
    console.error(error);
  }
}

(async () => {
  for await (const value of errorHandlingGenerator()) {
    console.log(value); // 'Result 2' (после обработки ошибки)
  }
})();
```

# Ресурсы

Тип данных Symbol - [ТЫК](#)

Перебираемые объекты - [ТЫК](#)

Генераторы - [ТЫК](#)

Асинхронные итераторы и генераторы - [ТЫК](#)