

Раздел

[Генераторы, продвинутая итерация](#)

Навигация по уроку

Функция-генератор

Перебор генераторов

Использование генераторов для перебираемых объектов

Композиция генераторов

yield – дорога в обе стороны


generator.throw

Итого

Задачи (1)

Комментарии

Поделиться

[Редактировать на GitHub](#)[🏠 → Язык программирования JavaScript](#)
[→ Генераторы, продвинутая итерация](#) 30-го ноября 2019

Генераторы

Обычные функции возвращают только одно-единственное значение (или ничего).

Генераторы могут порождать (yield) множество значений одно за другим, по мере необходимости. Генераторы отлично работают с перебираемыми объектами и позволяют легко создавать потоки данных.

Функция-генератор

Для объявления генератора используется специальная синтаксическая конструкция: `function*`, которая называется «функция-генератор».

Выглядит она так:

```
1 function* generateSequence() {  
2   yield 1;  
3   yield 2;  
4   return 3;  
5 }
```

Функции-генераторы ведут себя не так, как обычные. Когда такая функция вызвана, она не выполняет свой код. Вместо этого она возвращает специальный объект, так называемый «генератор», для управления её выполнением.




Вот, посмотрите:



```
1 function* generateSequence() {  
2   yield 1;  
3   yield 2;  
4   return 3;  
5 }  
6  
7 // "функция-генератор" создаёт объект "генератор"  
8 let generator = generateSequence();  
9 alert(generator); // [object Generator]
```

Выполнение кода функции ещё не началось:

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



Основным методом генератора является `next()`. При вызове он запускает выполнение кода до ближайшей инструкции `yield <значение>` (значение может отсутствовать, в этом случае оно предполагается равным `undefined`). По достижении `yield` выполнение функции приостанавливается, а соответствующее значение – возвращается во внешний код:

Результатом метода `next()` всегда является объект с двумя свойствами:

- `value`: значение из `yield`.
- `done`: `true`, если выполнение функции завершено, иначе `false`.

Например, здесь мы создаём генератор и получаем первое из возвращаемых им значений:

```
1 function* generateSequence() {
2   yield 1;
3   yield 2;
4   return 3;
5 }
6
7 let generator = generateSequence();
8
9 let one = generator.next();
10
11 alert(JSON.stringify(one)); // {value: 1, done: false}
```

На данный момент мы получили только первое значение, выполнение функции остановлено на второй строке:

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

← {value: 1, done: false}

Повторный вызов `generator.next()` возобновит выполнение кода и вернёт результат следующего `yield`:

```
1 let two = generator.next();
2
3 alert(JSON.stringify(two)); // {value: 2, done: false}
```

< >

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

← {value: 2, done: false}

И, наконец, последний вызов завершит выполнение функции и вернёт результат `return`:

```
1 let three = generator.next();
2
3 alert(JSON.stringify(three)); // {value: 3, done: true}
```

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

→ {value: 3, done: true}

Сейчас генератор полностью выполнен. Мы можем увидеть это по свойству `done:true` и обработать `value:3` как окончательный результат.

Новые вызовы `generator.next()` больше не имеют смысла. Впрочем, если они и будут, то не вызовут ошибки, но будут возвращать один и тот же объект: `{done: true}`.

Раздел

Генераторы, продвинутая итерация

Навигация по уроку

Функция-генератор

Перебор генераторов

Использование генераторов для перебираемых объектов

Композиция генераторов

`yield` – дорога в обе стороны

`generator.throw`

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub

Раздел

Генераторы, продвинутая итерация

Навигация по уроку

Функция-генератор

Перебор генераторов

Использование генераторов для перебираемых объектов

Композиция генераторов

yield – дорога в обе стороны

generator.throw

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



function* f(...) или function *f(...)?

Нет разницы, оба синтаксиса корректны.

Но обычно предпочтителен первый вариант, так как звёздочка относится к типу объявляемой сущности (function* – «функция-генератор»), а не к её названию, так что резонно расположить её у слова function.

Перебор генераторов

Как вы, наверное, уже догадались по наличию метода next(), генераторы являются **перебираемыми** объектами.

Возвращаемые ими значения можно перебирать через for...of:

```
1 function* generateSequence() {
2   yield 1;
3   yield 2;
4   return 3;
5 }
6
7 let generator = generateSequence();
8
9 for(let value of generator) {
10  alert(value); // 1, затем 2
11 }
```

Выглядит гораздо красивее, чем использование .next().value, верно?

...Но обратите внимание: пример выше выводит значение 1, затем 2. Значение 3 выведено не будет!

Это из-за того, что перебор через for...of игнорирует последнее значение, при котором done: true. Поэтому, если мы хотим, чтобы были все значения при переборе через for...of, то надо возвращать их через yield:

```
1 function* generateSequence() {
2   yield 1;
3   yield 2;
4   yield 3;
5 }
6
7 let generator = generateSequence();
8
9 for(let value of generator) {
10  alert(value); // 1, затем 2, затем 3
11 }
```

Так как генераторы являются перебираемыми объектами, мы можем использовать всю связанную с ними функциональность, например оператор расширения ...:

```
1 function* generateSequence() {
2   yield 1;
3   yield 2;
4   yield 3;
5 }
6
7 let sequence = [0, ...generateSequence()];
8
9 alert(sequence); // 0, 1, 2, 3
```

В коде выше ...generateSequence() превращает перебираемый объект-генератор в массив элементов (подробнее ознакомиться с оператором

Раздел

[Генераторы, продвинутая итерация](#)

Навигация по уроку

Функция-генератор

Перебор генераторов

Использование генераторов для перебираемых объектов

Композиция генераторов

yield – дорога в обе стороны

generator.throw

Итого

Задачи (1)

Комментарии

Поделиться

[Редактировать на GitHub](#)

Использование генераторов для перебираемых объектов

Некоторое время назад, в главе [Перебираемые объекты](#), мы создали перебираемый объект `range`, который возвращает значения `from..to`.

Давайте вспомним код:

```

1  let range = {
2    from: 1,
3    to: 5,
4
5    // for..of range вызывает этот метод один раз в самом
6    [Symbol.iterator]() {
7      // ..он возвращает перебираемый объект:
8      // далее for..of работает только с этим объектом, з
9      return {
10         current: this.from,
11         last: this.to,
12
13         // next() вызывается при каждой итерации цикла fo
14         next() {
15           // нужно вернуть значение как объект {done:...,
16           if (this.current <= this.last) {
17             return { done: false, value: this.current++ }
18           } else {
19             return { done: true };
20           }
21         }
22       };
23     }
24   };
25
26   // при переборе объекта range будут выведены числа от г
27   alert([...range]); // 1,2,3,4,5

```

Мы можем использовать функцию-генератор для итерации, указав её в `Symbol.iterator`.

Вот тот же `range`, но с гораздо более компактным итератором:

```

1  let range = {
2    from: 1,
3    to: 5,
4
5    *[Symbol.iterator]() { // краткая запись для [Symbol.
6      for(let value = this.from; value <= this.to; value+
7        yield value;
8      }
9    }
10 };
11
12 alert( [...range] ); // 1,2,3,4,5

```

Это работает, потому что `range[Symbol.iterator]()` теперь возвращает генератор, и его методы – в точности то, что ожидает `for..of`:

- у него есть метод `.next()`
- который возвращает значения в виде `{value: ..., done: true/false}`

Это не совпадение, конечно. Генераторы были добавлены в язык JavaScript, в частности, с целью упростить создание перебираемых объектов.

Вариант с генератором намного короче, чем исходный вариант перебираемого `range`, и сохраняет те же функциональные возможности.

Раздел

Генераторы, продвинутая итерация

Навигация по уроку

Функция-генератор

Перебор генераторов

Использование генераторов для перебираемых объектов

Композиция генераторов

yield – дорога в обе стороны

generator.throw

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



Генераторы могут генерировать бесконечно

В примерах выше мы генерировали конечные последовательности, но мы также можем сделать генератор, который будет возвращать значения бесконечно. Например, бесконечная последовательность псевдослучайных чисел.

Конечно, нам потребуется `break` (или `return`) в цикле `for..of` по такому генератору, иначе цикл будет продолжаться бесконечно, и скрипт «зависнет».

Композиция генераторов

Композиция генераторов – это особенная возможность генераторов, которая позволяет прозрачно «встраивать» генераторы друг в друга.

Например, у нас есть функция для генерации последовательности чисел:

```
1 function* generateSequence(start, end) {
2   for (let i = start; i <= end; i++) yield i;
3 }
```

Мы хотели бы использовать её при генерации более сложной последовательности:

- сначала цифры `0..9` (с кодами символов `48...57`)
- за которыми следуют буквы в верхнем регистре `A..Z` (коды символов `65...90`)
- за которыми следуют буквы алфавита `a..z` (коды символов `97...122`)

Мы можем использовать такую последовательность для генерации паролей, выбирать символы из неё (может быть, ещё добавить символы пунктуации), но сначала её нужно сгенерировать.

В обычной функции, чтобы объединить результаты из нескольких других функций, мы вызываем их, сохраняем промежуточные результаты, а затем в конце их объединяем.

Для генераторов есть особый синтаксис `yield*`, который позволяет «вкладывать» генераторы один в другой (осуществлять их композицию).

Вот генератор с композицией:

```
1 function* generateSequence(start, end) {
2   for (let i = start; i <= end; i++) yield i;
3 }
4
5 function* generatePasswordCodes() {
6
7   // 0..9
8   yield* generateSequence(48, 57);
9
10  // A..Z
11  yield* generateSequence(65, 90);
12
13  // a..z
14  yield* generateSequence(97, 122);
15 }
16
17
18 let str = '';
19
20 for(let code of generatePasswordCodes()) {
21   str += String.fromCharCode(code);
22 }
23
24 alert(str); // 0..9A..Za..z
```



Раздел

Генераторы, продвинутая итерация

Навигация по уроку

Функция-генератор

Перебор генераторов

Использование генераторов для перебираемых объектов

Композиция генераторов

yield – дорога в обе стороны

generator.throw

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



Директива `yield*` делегирует выполнение другому генератору. Этот термин означает, что `yield* gen` перебирает генератор `gen` и прозрачно направляет его вывод наружу. Как если бы значения были сгенерированы внешним генератором.

Результат – такой же, как если бы мы встроили код из вложенных генераторов:

```
1 function* generateSequence(start, end) {
2   for (let i = start; i <= end; i++) yield i;
3 }
4
5 function* generateAlphaNum() {
6
7   // yield* generateSequence(48, 57);
8   for (let i = 48; i <= 57; i++) yield i;
9
10  // yield* generateSequence(65, 90);
11  for (let i = 65; i <= 90; i++) yield i;
12
13  // yield* generateSequence(97, 122);
14  for (let i = 97; i <= 122; i++) yield i;
15
16 }
17
18 let str = '';
19
20 for(let code of generateAlphaNum()) {
21   str += String.fromCharCode(code);
22 }
23
24 alert(str); // 0..9a..zA..Z
```



Композиция генераторов – естественный способ вставлять вывод одного генератора в поток другого. Она не использует дополнительную память для хранения промежуточных результатов.



yield – дорога в обе стороны

До этого момента генераторы сильно напоминали перебираемые объекты, со специальным синтаксисом для генерации значений. Но на самом деле они намного мощнее и гибче.

Всё дело в том, что `yield` – дорога в обе стороны: он не только возвращает результат наружу, но и может передавать значение извне в генератор.

Чтобы это сделать, нам нужно вызвать `generator.next(arg)` с аргументом. Этот аргумент становится результатом `yield`.

Продemonстрируем это на примере:

```
1 function* gen() {
2   // Передаём вопрос во внешний код и ожидаем ответа
3   let result = yield "2 + 2 = ?"; // (*)
4
5   alert(result);
6 }
7
8 let generator = gen();
9
10 let question = generator.next().value; // <-- yield воз
11
12 generator.next(4); // --> передаём результат в генератор
```



Генератор

Вызывающий код

```
function* gen() {
  let result = yield "2 + 2 = ?";
}
```

→ question = "2 + 2 = ?"

Раздел

Генераторы, продвинутая итерация

Навигация по уроку

Функция-генератор

Перебор генераторов

Использование генераторов для перебираемых объектов

Композиция генераторов

yield – дорога в обе стороны

generator.throw

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



```
    alert(result); // 4
  }
}
.next(4)
```

1. Первый вызов `generator.next()` – всегда без аргумента, он начинает выполнение и возвращает результат первого `yield` `"2+2=?"`. На этой точке генератор приостанавливает выполнение.
2. Затем, как показано на картинке выше, результат `yield` переходит во внешний код в переменную `question`.
3. При `generator.next(4)` выполнение генератора возобновляется, а 4 выходит из присваивания как результат: `let result = 4`.

Обратите внимание, что внешний код не обязан немедленно вызывать `next(4)`. Ему может потребоваться время. Это не проблема, генератор подождёт.

Например:

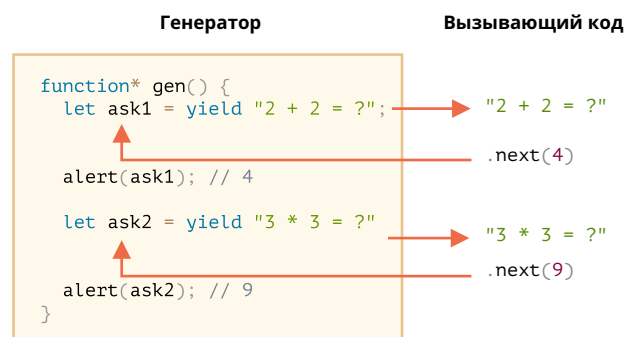
```
1 // возобновить генератор через некоторое время
2 setTimeout(() => generator.next(4), 1000);
```

Как видно, в отличие от обычных функций, генератор может обмениваться результатами с вызывающим кодом, передавая значения в `next/yield`.

Чтобы сделать происходящее более очевидным, вот ещё один пример с большим количеством вызовов:

```
1 function* gen() {
2   let ask1 = yield "2 + 2 = ?";
3
4   alert(ask1); // 4
5
6   let ask2 = yield "3 * 3 = ?";
7
8   alert(ask2); // 9
9 }
10
11 let generator = gen();
12
13 alert( generator.next().value ); // "2 + 2 = ?"
14
15 alert( generator.next(4).value ); // "3 * 3 = ?"
16
17 alert( generator.next(9).done ); // true
```

Картинка выполнения:



1. Первый `.next()` начинает выполнение... Оно доходит до первого `yield`.
2. Результат возвращается во внешний код.
3. Второй `.next(4)` передаёт 4 обратно в генератор как результат первого `yield` и возобновляет выполнение.

Раздел

[Генераторы, продвинутая итерация](#)

Навигация по уроку

Функция-генератор

Перебор генераторов

Использование генераторов для перебираемых объектов

Композиция генераторов

yield – дорога в обе стороны

generator.throw

Итого

Задачи (1)

Комментарии

Поделиться



[Редактировать на GitHub](#)



4. ...Оно доходит до второго `yield`, который станет результатом `.next(4)`.
5. Третий `next(9)` передаёт 9 в генератор как результат второго `yield` и возобновляет выполнение, которое завершается окончанием функции, так что `done: true`.

Получается такой «пинг-понг»: каждый `next(value)` передаёт в генератор значение, которое становится результатом текущего `yield`, возобновляет выполнение и получает выражение из следующего `yield`.

generator.throw

Как мы видели в примерах выше, внешний код может передавать значение в генератор как результат `yield`.

...Но можно передать не только результат, но и инициировать ошибку. Это естественно, так как ошибка является своего рода результатом.

Для того, чтобы передать ошибку в `yield`, нам нужно вызвать `generator.throw(err)`. В таком случае исключение `err` возникнет на строке с `yield`.

Например, здесь `yield "2 + 2 = ?"` приведёт к ошибке:

```
1 function* gen() {
2   try {
3     let result = yield "2 + 2 = ?"; // (1)
4
5     alert("Выполнение программы не дойдёт до этой строк
6   } catch(e) {
7     alert(e); // покажет ошибку
8   }
9 }
10
11 let generator = gen();
12
13 let question = generator.next().value;
14
15 generator.throw(new Error("Ответ не найден в моей базе ,
```

Ошибка, которая проброшена в генератор на строке (2), приводит к исключению на строке (1) с `yield`. В примере выше `try..catch` перехватывает её и отображает.

Если мы не хотим перехватывать её, то она, как и любое обычное исключение, «вывалится» из генератора во внешний код.

Текущая строка вызывающего кода – это строка с `generator.throw`, отмечена (2). Таким образом, мы можем отловить её во внешнем коде, как здесь:

```
1 function* generate() {
2   let result = yield "2 + 2 = ?"; // Ошибка в этой стро
3 }
4
5 let generator = generate();
6
7 let question = generator.next().value;
8
9 try {
10  generator.throw(new Error("Ответ не найден в моей баз
11 } catch(e) {
12  alert(e); // покажет ошибку
13 }
```

Если же ошибка и там не перехвачена, то дальше – как обычно, она выпадает наружу и, если не перехвачена, «повалит» скрипт.

Итого

Раздел

Генераторы, продвинутая итерация

Навигация по уроку

Функция-генератор

Перебор генераторов

Использование генераторов для перебираемых объектов

Композиция генераторов

yield – дорога в обе стороны

generator.throw

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



- Генераторы создаются при помощи функций-генераторов `function* f(...) {...}`.
- Внутри генераторов и только внутри них существует оператор `yield`.
- Внешний код и генератор обмениваются промежуточными результатами посредством вызовов `next/yield`.

В современном JavaScript генераторы используются редко. Но иногда они оказываются полезными, потому что способность функции обмениваться данными с вызывающим кодом во время выполнения совершенно уникальна. И, конечно, для создания перебираемых объектов.

Также, в следующей главе мы будем изучать асинхронные генераторы, которые используются, чтобы читать потоки асинхронно сгенерированных данных (например, постранично загружаемые из сети) в цикле `for await ... of`.

В веб-программировании мы часто работаем с потоками данных, так что это ещё один важный случай использования.

✓ Задачи

Псевдослучайный генератор

Есть много областей, где нам нужны случайные данные.

Одной из них является тестирование. Нам могут понадобиться случайные данные: текст, числа и т.д., чтобы хорошо всё проверить.

В JavaScript мы можем использовать `Math.random()`. Но если что-то пойдёт не так, то нам нужно будет перезапустить тест, используя те же самые данные.

Для этого используются так называемые «сеяные псевдослучайные генераторы». Они получают «зерно», как первое значение, и затем генерируют следующее, используя формулу. Так что одно и то же зерно даёт одинаковую последовательность, и, следовательно, весь поток легко воспроизводим. Нам нужно только запомнить зерно, чтобы воспроизвести последовательность.

Пример такой формулы, которая генерирует более-менее равномерно распределённые значения:

```
1 next = previous * 16807 % 2147483647
```

Если мы используем 1 как зерно, то значения будут:

1. 16807
2. 282475249
3. 1622650073
4. ...и так далее...

Задачей является создать функцию-генератор `pseudoRandom(seed)`, которая получает `seed` и создаёт генератор с указанной формулой.

Пример использования:

```
1 let generator = pseudoRandom(1);
2
3 alert(generator.next().value); // 16807
4 alert(generator.next().value); // 282475249
5 alert(generator.next().value); // 1622650073
```

[Открыть песочницу с тестами для задачи.](#)

решение

Раздел

[Генераторы, продвинутая итерация](#)

Навигация по уроку

Функция-генератор

Перебор генераторов

Использование генераторов для перебираемых объектов

Композиция генераторов

yield – дорога в обе стороны

generator.throw

Итого

Задачи (1)

Комментарии

Поделиться



[Редактировать на GitHub](#)



Комментарии

перед тем как писать...



© 2007–2020 Илья Кантор | [о проекте](#) | [связаться с нами](#) | [пользовательское соглашение](#) | [политика конфи](#)

