Асинхронные итераторы и генераторы

Асинхронные итераторы позволяют перебирать данные, поступающие асинхронно. Например, когда мы загружаем что-то по частям по сети. Асинхронные генераторы делают такой перебор ещё удобнее.

Давайте сначала рассмотрим простой пример, чтобы понять синтаксис, а затем – реальный практический.

Асинхронные итераторы

Асинхронные итераторы похожи на обычные итераторы, но имеют некоторые синтаксические отличия.

«Обычный» перебираемый объект, как подробно рассказано в главе Перебираемые объекты, выглядит примерно так:

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

  // for..of вызывает этот метод один раз в самом начал
  [Symbol.iterator]() {
    // ...возвращает объект-итератор:
    // далее for..of работает только с этим объектом, з
    return {
      current: this.from,
      last: this.to,

      // next() вызывается на каждой итерации цикла for
      next() { // (2)
        // должен возвращать значение в виде объекта {d
        if (this.current <= this.last) {
          return { done: false, value: this.current++ }
        } else {
          return { done: true };
        }
      }
    };
  }
};

for(let value of range) {
  alert(value); // 1, потом 2, потом 3, потом 4, потом
}
```

Если нужно, пожалуйста, ознакомьтесь с главой про итераторы, где обычные итераторы разбираются подробно.

Чтобы сделать объект итерируемым асинхронно:

1. Используется `Symbol.asyncIterator` вместо `Symbol.iterator`.
2. `next()` должен возвращать промис.
3. Чтобы перебрать такой объект, используется цикл `for await (let item of iterable)`.

Давайте создадим итерируемый объект `range`, как и в предыдущем примере, но теперь он будет возвращать значения асинхронно, по одному в секунду:

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

  // for await..of вызывает этот метод один раз в самом
  [Symbol.asyncIterator]() { // (1)
    // ...возвращает объект-итератор:
    // далее for await..of работает только с этим объек
    // запрашивая у него следующие значения вызовом nex
    return {
      current: this.from,
      last: this.to,

      // next() вызывается на каждой итерации цикла for
      async next() { // (2)
        // должен возвращать значение как объект {done:
        // (автоматически оборачивается в промис с помо

        // можно использовать await внутри для асинхрон
        await new Promise(resolve => setTimeout(resolve

        if (this.current <= this.last) {
          return { done: false, value: this.current++ }
        } else {
          return { done: true };
        }
      }
    };
  }
};

(async () => {

  for await (let value of range) { // (4)
    alert(value); // 1,2,3,4,5
  }

})()
```

Как видим, структура похожа на обычные итераторы:

1. Чтобы сделать объект асинхронно итерируемым, он должен иметь метод `Symbol.asyncIterator` `(1)`.
2. Этот метод должен возвращать объект с методом `next()`, который в свою очередь возвращает промис `(2)`.
3. Метод `next()` не обязательно должен быть `async`, он может быть обычным методом, возвращающим промис, но `async` позволяет использовать `await`, так что это удобно. Здесь мы просто делаем паузу на одну секунду `(3)`.
4. Для итерации мы используем `for await (let value of range)` `(4)`, добавляя «await» после «for». Он вызовет `range[Symbol.asyncIterator]()` один раз, а затем его метод `next()` для получения значений.

Вот небольшая шпаргалка:

	Итераторы	Асинхронные итераторы
Метод для создания итерируемого объекта	`Symbol.iterator`	`Symbol.asyncIterator`
`next()` возвращает	любое значение	промис
для цикла используйте	`for..of`	`for await..of`

Оператор расширения `...` не работает асинхронно

Функции, которые требуют обычных синхронных итераторов, не работают с асинхронными.

Например, оператор расширения (три точки `...`) не будет работать:

```
alert( [...range] ); // Ошибка, нет Symbol.iterator
```

Это естественно, так как он ожидает `Symbol.iterator`, как и `for..of` без `await`. Ему не подходит `Symbol.asyncIterator`.

Асинхронные генераторы

Как мы уже знаем, в JavaScript есть генераторы, и они являются перебираемыми.

Давайте вспомним генератор последовательности из главы Генераторы. Он генерирует последовательность значений от `start` до `end`:

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

for(let value of generateSequence(1, 5)) {
  alert(value); // 1, потом 2, потом 3, потом 4, потом
}
```

В обычных генераторах мы не можем использовать `await`. Все значения должны поступать синхронно: в `for..of` нет места для задержки, это синхронная конструкция.

Но что если нам нужно использовать `await` в теле генератора? Для выполнения сетевых запросов, например.

Нет проблем, просто добавьте в начале `async`, например, вот так:

```
async function* generateSequence(start, end) {

  for (let i = start; i <= end; i++) {

    // ура, можно использовать await!
    await new Promise(resolve => setTimeout(resolve, 10

    yield i;
  }

}

(async () => {

  let generator = generateSequence(1, 5);
  for await (let value of generator) {
    alert(value); // 1, потом 2, потом 3, потом 4, пото
  }

})();
```

Теперь у нас есть асинхронный генератор, который можно перебирать с помощью `for await ... of`.

Это действительно очень просто. Мы добавляем ключевое слово `async`, и внутри генератора теперь можно использовать `await`, а также промисы и другие асинхронные функции.

С технической точки зрения, ещё одно отличие асинхронного генератора заключается в том, что его метод `generator.next()` теперь тоже асинхронный и возвращает промисы.

Из обычного генератора мы можем получить значения при помощи `result = generator.next()`. Для асинхронного нужно добавить `await`, вот так:

```
result = await generator.next(); // result = {value: ..
```

Асинхронно перебираемые объекты

Как мы уже знаем, чтобы сделать объект перебираемым, нужно добавить к нему `Symbol.iterator`.

```
let range = {
  from: 1,
  to: 5,
  [Symbol.iterator]() {
    return <объект с next, чтобы сделать range перебира
  }
}
```

Обычная практика для `Symbol.iterator` – возвращать генератор, а не простой объект с `next`, как в предыдущем примере.

Давайте вспомним пример из главы Генераторы:

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

  *[Symbol.iterator]() { // сокращение для [Symbol.iter
    for(let value = this.from; value <= this.to; value+
      yield value;
    }
  }
};

for(let value of range) {
  alert(value); // 1, потом 2, потом 3, потом 4, потом
}
```

Здесь созданный объект `range` является перебираемым, а генератор `*[Symbol.iterator]` реализует логику для перечисления значений.

Если хотим добавить асинхронные действия в генератор, нужно заменить `Symbol.iterator` на асинхронный `Symbol.asyncIterator`:

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

  async *[Symbol.asyncIterator]() { // то же, что и [Sy
    for(let value = this.from; value <= this.to; value+

      // пауза между значениями, ожидание
      await new Promise(resolve => setTimeout(resolve,

      yield value;
    }
  }
};

(async () => {

  for await (let value of range) {
    alert(value); // 1, потом 2, потом 3, потом 4, пото
```

```
20    }
21
22  })();
```

Теперь значения поступают с задержкой в одну секунду между ними.

Пример из реальной практики

До сих пор мы видели простые примеры, чтобы просто получить базовое представление. Теперь давайте рассмотрим реальную ситуацию.

Есть много онлайн-сервисов, которые предоставляют данные постранично. Например, когда нам нужен список пользователей, запрос возвращает предопределённое количество (например, 100) пользователей – «одну страницу», и URL следующей страницы.

Этот подход очень распространён, и речь не только о пользователях, а о чём угодно. Например, GitHub позволяет получать коммиты таким образом, с разбивкой по страницам:

- Нужно сделать запрос на URL в виде `https://api.github.com/repos/<repo>/commits`.
- В ответ придёт JSON с 30 коммитами, а также со ссылкой на следующую страницу в заголовке `Link`.
- Затем можно использовать эту ссылку для следующего запроса, чтобы получить дополнительную порцию коммитов, и так далее.

Но нам бы, конечно же, хотелось вместо этого сложного взаимодействия иметь просто объект с коммитами, которые можно перебирать, вот так:

```
1  let repo = 'javascript-tutorial/en.javascript.info'; //
2
3  for await (let commit of fetchCommits(repo)) {
4    // обработка коммитов
5  }
```

Мы бы хотели сделать функцию `fetchCommits(repo)`, которая будет получать коммиты, делая запросы всякий раз, когда это необходимо. И пусть она сама разбирается со всем, что касается нумерации страниц, для нас это будет просто `for await..of`.

С асинхронными генераторами это довольно легко реализовать:

```
1  async function* fetchCommits(repo) {
2    let url = `https://api.github.com/repos/${repo}/commi
3
4    while (url) {
5      const response = await fetch(url, { // (1)
6        headers: {'User-Agent': 'Our script'}, // GitHub
7      });
8
9      const body = await response.json(); // (2) ответ в
10
11     // (3) Ссылка на следующую страницу находится в заг
12     let nextPage = response.headers.get('Link').match(/
13     nextPage = nextPage && nextPage[1];
14
15     url = nextPage;
16
17     for(let commit of body) { // (4) вернуть коммиты од
18       yield commit;
19     }
20   }
21 }
```

1. Мы используем метод fetch браузера для загрузки с удалённого URL. Он позволяет при необходимости добавлять авторизацию и другие заголовки, здесь GitHub требует `User-Agent`.

2. Результат `fetch` обрабатывается как JSON, это опять-таки метод, присущий `fetch`.
3. Нужно получить URL следующей страницы из заголовка ответа `Link`. Он имеет специальный формат, поэтому мы используем регулярное выражение. URL следующей страницы может выглядеть как `https://api.github.com/repositories/93253246/commits?page=2`, он генерируется самим GitHub.
4. Затем мы выдаём все полученные коммиты, а когда они закончатся – сработает следующая итерация `while(url)`, которая сделает ещё один запрос.

Пример использования (показывает авторов коммитов в консоли):

```
(async () => {

  let count = 0;

  for await (const commit of fetchCommits('javascript-t

    console.log(commit.author.login);

    if (++count == 100) { // остановимся на 100 коммита
      break;
    }
  }

})();
```

Это именно то, что мы хотели. Внутренняя механика постраничных запросов снаружи не видна. Для нас это просто асинхронный генератор, который возвращает коммиты.

Итого

Обычные итераторы и генераторы прекрасно работают с данными, которые не требуют времени для их создания или получения.

Когда мы ожидаем, что данные будут поступать асинхронно, с задержками, можно использовать их асинхронные аналоги и `for await..of` вместо `for..of`.

Синтаксические различия между асинхронными и обычными итераторами:

	Перебираемый объект	Асинхронно перебираемый
Метод для получения итератора	`Symbol.iterator`	`Symbol.asyncIterator`
`next()` возвращает	`{value:…, done: true/false}`	промис, который завершается с `{value:…, done: true/false}`

Синтаксические различия между асинхронными и обычными генераторами:

	Генераторы	Асинхронные генераторы
Объявление	`function*`	`async function*`
`generator.next()` возвращает	`{value:…, done: true/false}`	промис, который завершается с `{value:…, done: true/false}`

В веб-разработке мы часто встречаемся с потоками данных, когда они поступают по частям. Например, загрузка или выгрузка большого файла.

Мы можем использовать асинхронные генераторы для обработки таких данных. Также заметим, что в некоторых окружениях, например, браузерах, есть и другое API, называемое Streams (потоки), который предоставляет специальные интерфейсы для работы с такими потоками данных, их

преобразования и передачи из одного потока в другой (например, загрузка из одного источника и сразу отправка в другое место).

Комментарии

перед тем как писать...