7th February 2020

Async iterators and generators

Asynchronous iterators allow to iterate over data that comes asynchronously, on-demand. For instance, when we download something chunk-by-chunk over a network. Asynchronous generators make it even more convenient.

Let's see a simple example first, to grasp the syntax, and then review a real-life use case.

Async iterators

Asynchronous iterators are similar to regular iterators, with a few syntactic differences.

A "regular" iterable object, as described in the chapter Iterables, looks like this:

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

  // for..of calls this method once in the very beginning
  [Symbol.iterator]() {
    // ...it returns the iterator object:
    // onward, for..of works only with that object,
    // asking it for next values using next()
    return {
      current: this.from,
      last: this.to,

      // next() is called on each iteration by the for..of loop
      next() { // (2)
        // it should return the value as an object {done:.., value :...}
        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      }
    };
  }
};

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

If necessary, please refer to the chapter about iterables for details about regular iterators.

To make the object iterable asynchronously:

1. We need to use `Symbol.asyncIterator` instead of `Symbol.iterator`.
2. `next()` should return a promise.
3. To iterate over such an object, we should use `for await (let item of iterable)` loop.

Let's make an iterable `range` object, like the one before, but now it will return values asynchronously, one per second:

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

  // for await..of calls this method once in the very beginning
  [Symbol.asyncIterator]() { // (1)
    // ...it returns the iterator object:
    // onward, for await..of works only with that object,
    // asking it for next values using next()
    return {
      current: this.from,
      last: this.to,

      // next() is called on each iteration by the for await..of loop
      async next() { // (2)
        // it should return the value as an object {done:.., value :...}
        // (automatically wrapped into a promise by async)

        // can use await inside, do async stuff:
        await new Promise(resolve => setTimeout(resolve, 1000)); // (3)

        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
  }

})()
```

As we can see, the structure is similar to regular iterators:

1. To make an object asynchronously iterable, it must have a method `Symbol.asyncIterator` `(1)`.
2. This method must return the object with `next()` method returning a promise `(2)`.
3. The `next()` method doesn't have to be `async`, it may be a regular method returning a promise, but `async` allows to use `await`, so that's convenient. Here we just delay for a second `(3)`.
4. To iterate, we use `for await(let value of range)` `(4)`, namely add "await" after "for". It calls `range[Symbol.asyncIterator]()` once, and then its `next()` for values.

Here’s a small cheatsheet:

	Iterators	Async iterators
Object method to provide iterator	`Symbol.iterator`	`Symbol.asyncIterator`
`next()` return value is	any value	`Promise`
to loop, use	`for..of`	`for await..of`

The spread syntax `...` doesn’t work asynchronously

Features that require regular, synchronous iterators, don’t work with asynchronous ones.

For instance, a spread syntax won’t work:

```
alert( [...range] ); // Error, no Symbol.iterator
```

That’s natural, as it expects to find `Symbol.iterator`, same as `for..of` without `await`. Not `Symbol.asyncIterator`.

Async generators

As we already know, JavaScript also supports generators, and they are iterable.

Let’s recall a sequence generator from the chapter Generators. It generates a sequence of values from `start` to `end`:

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

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

In regular generators we can’t use `await`. All values must come synchronously: there’s no place for delay in `for..of`, it’s a synchronous construct.

But what if we need to use `await` in the generator body? To perform network requests, for instance.

No problem, just prepend it with `async`, like this:

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

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

    // yay, can use await!
    await new Promise(resolve => setTimeout(resolve, 1000));

    yield i;
  }

}

(async () => {

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

})();
```

Now we have the async generator, iterable with `for await...of`.

It’s indeed very simple. We add the `async` keyword, and the generator now can use `await` inside of it, rely on promises and other async functions.

Technically, another difference of an async generator is that its `generator.next()` method is now asynchronous also, it returns promises.

In a regular generator we’d use `result = generator.next()` to get values. In an async generator, we should add `await`, like this:

```
result = await generator.next(); // result = {value: ..., done: true/false}
```

Async iterables

As we already know, to make an object iterable, we should add `Symbol.iterator` to it.

```
let range = {
  from: 1,
  to: 5,
  [Symbol.iterator]() {
    return <object with next to make range iterable>
  }
}
```

A common practice for `Symbol.iterator` is to return a generator, rather than a plain object with `next` as in the example before.

Let’s recall an example from the chapter Generators:

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

  *[Symbol.iterator]() { // a shorthand for [Symbol.iterator]: function*()
    for(let value = this.from; value <= this.to; value++) {
      yield value;
    }
  }
};

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

Here a custom object `range` is iterable, and the generator `*[Symbol.iterator]` implements the logic for listing values.

If we'd like to add async actions into the generator, then we should replace `Symbol.iterator` with async `Symbol.asyncIterator`:

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

  async *[Symbol.asyncIterator]() { // same as [Symbol.asyncIterator]: async
    for(let value = this.from; value <= this.to; value++) {

      // make a pause between values, wait for something
      await new Promise(resolve => setTimeout(resolve, 1000));

      yield value;
    }
  }
};

(async () => {

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

})();
```

Now values come with a delay of 1 second between them.

Real-life example

So far we've seen simple examples, to gain basic understanding. Now let's review a real-life use case.

There are many online services that deliver paginated data. For instance, when we need a list of users, a request returns a pre-defined count (e.g. 100 users) – "one page", and provides a URL to the next page.

This pattern is very common. It's not about users, but just about anything. For instance, GitHub allows to retrieve commits in the same, paginated fashion:

- We should make a request to URL in the form `https://api.github.com/repos/<repo>/commits` .
- It responds with a JSON of 30 commits, and also provides a link to the next page in the `Link` header.
- Then we can use that link for the next request, to get more commits, and so on.

But we'd like to have a simpler API: an iterable object with commits, so that we could go over them like this:

```
let repo = 'javascript-tutorial/en.javascript.info'; // GitHub repository to

for await (let commit of fetchCommits(repo)) {
  // process commit
}
```

We'd like to make a function `fetchCommits(repo)` that gets commits for us, making requests whenever needed. And let it care about all pagination stuff. For us it'll be a simple `for await..of`.

With async generators that's pretty easy to implement:

```
async function* fetchCommits(repo) {
  let url = `https://api.github.com/repos/${repo}/commits`;

  while (url) {
    const response = await fetch(url, { // (1)
      headers: {'User-Agent': 'Our script'}, // github requires user-agent he
    });

    const body = await response.json(); // (2) response is JSON (array of com

    // (3) the URL of the next page is in the headers, extract it
    let nextPage = response.headers.get('Link').match(/<(.*?)>; rel="next"/);
    nextPage = nextPage && nextPage[1];

    url = nextPage;

    for(let commit of body) { // (4) yield commits one by one, until the page
      yield commit;
    }
  }
}
```

1. We use the browser fetch method to download from a remote URL. It allows us to supply authorization and other headers if needed – here GitHub requires `User-Agent`.
2. The fetch result is parsed as JSON. That's again a `fetch`-specific method.
3. We should get the next page URL from the `Link` header of the response. It has a special format, so we use a regexp for that. The next page URL may look like `https://api.github.com/repositories/93253246/commits?page=2`. It's generated by GitHub itself.
4. Then we yield all commits received, and when they finish, the next `while(url)` iteration will trigger, making one more request.

An example of use (shows commit authors in console):

```
(async () => {

  let count = 0;

  for await (const commit of fetchCommits('javascript-tutorial/en.javascrip

    console.log(commit.author.login);

    if (++count == 100) { // let's stop at 100 commits
      break;
    }
  }

})();
```

That's just what we wanted. The internal mechanics of paginated requests is invisible from the outside. For us it's just an async generator that returns commits.

Summary

Regular iterators and generators work fine with the data that doesn't take time to generate.

When we expect the data to come asynchronously, with delays, their async counterparts can be used, and `for await..of` instead of `for..of`.

Syntax differences between async and regular iterators:

	Iterable	Async Iterable
Method to provide iterator	`Symbol.iterator`	`Symbol.asyncIterator`
`next()` return value is	`{value:…, done: true/false}`	`Promise` that resolves to `{value:…, done: true/false}`

Syntax differences between async and regular generators:

	Generators	Async generators
Declaration	`function*`	`async function*`
`next()` return value is	`{value:…, done: true/false}`	`Promise` that resolves to `{value:…, done: true/false}`

In web-development we often meet streams of data, when it flows chunk-by-chunk. For instance, downloading or uploading a big file.

We can use async generators to process such data. It's also noteworthy that in some environments, like in browsers, there's also another API called Streams, that provides special interfaces to work with such streams, to transform the data and to pass it from one stream to another (e.g. download from one place and immediately send elsewhere).

Previous lesson Next lesson

Share Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)